A Brief Summary of the Core Features of the Rust Programming Language

Vinith Krishnan

Rust is a general-purpose, multi-paradigm programming language developed by Mozilla research. It was initially published in 2010 with a first stable version appearing in May, 2015. It specifically targets the domain currently dominated by C and C++, namely systems programming, but differentiates itself by having been developed to prevent some of the problems related to invalid memory accesses (which generate segmentation faults).

Rust mainly focuses on these areas:-

- Move semantics
- Guaranteed memory safety.
- Threads without data races.
- Pattern matching.
- Type inference.
- Minimal runtime.
- Efficient C bindings.

Programming languages usually face trade-offs: you can have a language which is safe, but you give up control, or you can have a language with control, but it is unsafe. C++ falls into that latter category. While modern C++ is significantly safer than it used to be, there are fundamental aspects of C++ which make it impossible to ever be truly safe. Rust attempts to give you a language with 100 percent control, yet be absolutely safe. It does this through extensive compile-time checking, so you pay little and often no cost at runtime for its safety features.

1 Ownership

Rust implements a unique feature called ownership. This means that variables in Rust own the allocated data resources (such as vectors, stacks and other objects) they are bound to. This results in that every variable binding must own exactly one data resource.

For instance, suppose we have the code in Snippet 1. Variable 'p' allocates memory on the heap for an integer , and then the ownership moves from variable 'p' to variable 'q'. This is called move semantics in programming terms. After this, trying to use variable 'p' would result in a compile-error.

Snippet 1:-{
*l*et p = Box::new(5); // now owns a boxed int
let q = p; //ownership moves to q
println!(" ", p); //error
}

The same behavior goes with functions. If we pass a variable as argument to a function, the function will consume its ownership. This means that the code in Snippet 2 also gives a compile-error.

Snippet 2:-
{ fn main()
let m = Box::new(5);
function(m); // give away ownership
println!("The number is ", m); // error
}
// i takes over ownership
fn function(i: Box<i32>) {

}

Because of this, the ownership system alone would make programming difficult. Fortunately, you can also pass references to functions, and thus keeping the original ownership. This is known as borrowing in Rust.

It is also possible to copy data as opposed to moving them.This applies to all primitive data types, such as integers and floating point numbers. They implement a trait called Copy, which can be implemented for other data types as well. If we had not boxed the integer in Snippet 1, then the Copy trait would kick in and have variable q own a copy of the value, instead of the actual value. Thus, printing variable p would be a valid operation .

2 Manual memory management

Rust strives to achieve zero-cost abstractions, which its memory management is an excellent example of. Rust uses manual memory management in the sense that the programmer is in complete control of which memory is being allocated, just like in C. But unlike C, the programmers might not be aware of how much control they have because Rust automatically knows when to allocate and free memory. Memory management has been abstracted away from the programmers view, so they do not even notice it.

Manual memory management is possible due to the ownership feature. Whenever a variable goes out of scope, Rust frees the variable and all data it is bound to. This is the reason any data must be owned by exactly one variable.Let us see why. Suppose we have two variables owning the same data. If the first variable goes out of scope, both itself and the data it owned are automatically freed. Now, trying to use the second variable would be very dangerous because it owns data that does not exist anymore. This is why we would get a compile-error earlier in Snippet 1. As a consequence from ownership, it is always safe to free related data when their variable goes out of scope.

3 Type System

Rust uses affine types which aren't found in many other real-world languages, to ensure that there are no data-races, post-allocation memory accesses, etc.

Rust has three "realms" in which objects can be allocated: the stack, the local heap, and the exchange heap. These realms have corresponding pointer types: the borrowed pointer & T, the shared box @T, and the unique box T.

Boxes belong to a type called an affine type.The Rust compiler, at compile time, determines when the box comes in and goes out of scope, and inserts the appropriate calls there. Furthermore, boxes are a specific kind of affine type, known as a region.

```
{
  let x = box 5i;
}
```

This is similar to the following C-code:

```
{    int *x;
  x = (int *)malloc(sizeof(int));
  *x = 5;
  free(x);
}
```

Rust provides compile time guarantees through regions and Boxes. Regions and boxes is where ownership and borrowing comes from, which makes it impossible to

allocate the incorrect amount of memory, because Rust figures it out from the types.

One does not need to remember to free memory allocated, because Rust does it for you.

Rust ensures that this freeing happens at the right time, when it is truly not used. Usage after freeing is not possible. Rust enforces that no other writable pointers alias to this heap memory, which means writing to an invalid pointer is not possible.

Using boxes and references together is very common. For example:

```
fn add_one(x:: &int) -> int
{
  *x + 1
}

fn main()
{
  let x = box 5i;
  println!("{ }", add_one(&*x));
}
```

In this case, Rust knows that x is being 'borrowed' by the add_one() function, and since it's only reading the value, allows it.

References

[1] The Rust programming language: https://doc.rust-lang.org/book/ownership.html.

[2] The Rust Reference. https://doc.rust-lang.org/reference.html