Jonathan Davis and Anh Dao

Dr. Martha J. Kosa

CSC 4010 – 800

08 December, 2017

The Rust and Go Programming Languages

Rust is an open source systems programming language sponsored by the research division of Mozilla, a free-software community well known for the internet browser Firefox. The Rust programming language is statically typed and supports both the functional and procedural paradigms, with its procedural aspects being very similar to C++. The goal of the Rust Project is stated as being "To design and implement a safe, concurrent, practical systems language." ("Frequently Asked Questions") The creators of Rust wanted the language to have high concurrency and superior memory safety, while not sacrificing performance. Like Rust, Go is also an open source systems programming language. Go is developed by Google, who like Mozilla are also known for an internet browser, Chrome. Go is a strong, static typed language, and it follows the imperative paradigm. Google's goal in creating Go was to design a new programming language that would resolve common criticisms of other languages while maintaining their positive characteristics.

Rust's syntax is very similar to C++, with both languages using the equals sign for variable assignment, sharing many of the same control flow features, and necessitating semi colons to follow the end of each statement. However, Rust differs in several ways from C++ due to its support for the functional paradigm. Variable assignments in Rust require the "let" keyword, and the language also has keywords for pattern matching techniques found in most functional languages (The code for a “match” statement in Rust looks very much like a “switch” statement in C++). In Rust, almost every line in a function body is an expression. As such, a programmer does not need to explicitly specify what to return within a function; the evaluation of the last expression executed will be returned. This differs from C++, where a value to return from a function must explicitly be preceded with the “return” keyword.

Go’s syntax is very different from C++ because it inherits from C. However, Go’s syntax has many changes compared to C’s in order to keep the code as concise and readable as possible. A combination of declaration and initialization is used to let the programmer declare variables without specifying a name. This is due to the Type Inference feature of Go, a feature that is possible due to Go being a strong, statically typed language. A programmer just needs to use “:=” for variable initialization and “=” for reassigning a new value to a preexisting variable. This differs from C++, where a programmer must specify the type when declaring new variable. The use of a semi-colon to terminate statements is optional in Go because the end of a line also implies the termination of a statement. In C++, functions can only return one type of value, or no value at all (void). However, Go’s functions have the ability to return multiple variables of different types. The programmer simply need to make sure that the order of variable’s types returned match the return type’s declaration of the function.

In Rust, variable bindings have ownership of what they’re bound to, and the language makes sure that there is only one binding to any resource. Rust uses a system known as a borrow checker that handles ownership of resources. A variable cannot be used within a scope after the ownership of that variable’s binding has been moved. To overcome this, data types can either implement the Copy trait (traits are similar to interfaces in Java), or else references can be used with a preceding ampersand character, which is similar to C++. Also, neither Rust nor Go provide implicit type conversion between values, instead necessitating explicit type conversions. This differs from C++ and Java which support implicit type conversions, commonly in the form of widening conversion.

Rust achieves its memory safety mechanisms by utilizing lifetimes. Each reference, and anything that contains a reference, is tagged with a lifetime specifying the scope it's valid for. (“Lifetimes”) The memory safety features in place prevent a programmer from creating null and by extension dangling pointers, which is a nice safeguard that C++ does not offer. According to the Rust FAQ, “The only way to construct a value of type &Foo or &mut Foo is to specify an existing value of type Foo that the reference points to. The reference “borrows” the original value for a given region of code (the lifetime of the reference), and the value being borrowed from cannot be moved or destroyed for the duration of the borrow.” (“Frequently Asked Questions”). A programmer can replicate the testing of pointers as valid or invalid (what would be NULL) by using the Option type. The Option type can either be Some of a type (valid pointer) or None (invalid).

Rust does not have automatic garbage collection like other languages such as a Java or Go. Rust instead uses the Resource Acquisition Is Initialization, or RAII, principle. This means that whenever an object goes out of scope, its destructor is called and its owned resources are freed. As previously mentioned, Rust uses a system similar to Java interfaces called traits, which allow for ad hoc polymorphism. Like languages such as OCaml, Rust features a type inference system that will automatically figure out the types of variables depending on the context they are used in. Variables are immutable in Rust be default, and must be preceded with the keyword “mut” in their declaration in order to be changed after initialization.

Rust supports the use of generics, with usage resembling Java's generics or C++'s templates. The Go language does not support generics, instead using closures to accomplish similar functionality. A generic function in Rust can be type checked as soon as it is defined, although to achieve this the generic value must usually implement one or more prespecified traits. Rust uses traits and structs together to provide a sort of system that supports inheritance and polymorphism called implementations. Because Rust does not use class inheritance at all and only allows traits to inherit from other traits, mixins are able to easily be implemented and the diamond of death problem is prevented from occurring (which is also prevented in Java by disallowing multiple inheritance, but not prevented in C++ where multiple inheritance is legal).

Compared to object oriented paradigm language like Java, Go has embedding and interfaces features that replace traditional class inheritance. Interfaces provide runtime polymorphism, but in a limited form of structural typing rather than nominal typing. This is a core concept in Go's type system; instead of designing abstractions in terms of what kind of data types can hold, abstractions are designed in terms of what actions the types can execute. Go also allows type assertion and type switching. In Go there is a concept of an empty interface, denoted by "interface{}", that can refer to an item of any concrete type. This is equivalent to the base level object in Java, which is very important when dealing with static typing.

In the Go language, the total running time of a program always needs to be considered; this is where the concept of parallelism comes into play. Go has built-in libraries that support concurrency. The main construct of concurrency is a light-weight process called a goroutine. To create a goroutine, the go keyword must be used followed by a function invocation. The documentation on Go does not specify how goroutine is implemented, but operating system threads will get the multiplexed goroutines depending on the current implementation.

(“Concurrency”) Other than old school concurrency control schemes such as semaphores or mutex locks, a Go programmer will normally use channels, which are a simple way for two goroutines to communicate with one another and synchronize their execution. By default, message sends and receives will block until the other side is ready. (“Channels”) This allows goroutines to synchronize without explicit locks or condition variables. Channels can also be buffered for more precise uses. Message sends to a buffered channel block when the buffer is full and message receives block only when the buffer is empty. Even though concurrency is very useful in Go, the language does not provide any built-in notion of safe or verifiable concurrency.

In conclusion, there are several aspects of Rust and Go that are both similar and different between the two, as well as with commonly studied languages such as C++ and Java. Although Rust is a procedural and functional language, code at a glance looks strikingly similar to Go, an imperative language with no real functional features. Both Rust and Go were created to improve on the various aspects of programming languages that came before them, with Rust having the goal of having high concurrency and superior memory safety while not sacrificing performance, and Go having the goal to be a concurrent, garbage-collected language with fast compilation times. Both of these systems programming languages have promise to deliver new, innovative ways of solving problems that keep up with the everchanging computing landscape. (“The Go Memory Model”)

Works Cited

“Frequently Asked Questions.” *Frequently Asked Questions · The Rust Programming Language*, www.rust-lang.org/en-US/faq.html.

“Lifetimes.” *Rust Language Documentation*, doc.rust-lang.org/nomicon/lifetimes.html.

"The Go Memory Model". *Go Language Documentation*, https://golang.org/ref/mem

“Channels”. *A Tour of Go*, https://tour.golang.org/concurrency/2

“Concurrency”. *Go Resources*, https://www.golang-book.com/books/intro/10