A brief look at Rust

Creating robust software is not only a people, process and frameworks choice. The programming language and the ideas it is built upon play an important role on the quality and characteristics of the solution. A language is a lot more than its syntax. It is a combination of choices that ease software development through abstractions: the paradigms and language mechanisms that support that way of development, coding guarantees, supporting tools, community relationship, amount and quality of libraries, etc.

A Large-Scale Study of Programming Languages and Code Quality in GitHub concludes that: *The data indicates that functional languages are better than procedural languages; it suggests that disallowing implicit type conversion is better than allowing it; that static typing is better than dynamic; and that managed memory usage is better than unmanaged. Further, that the defect proneness of languages in general is not associated with software domains. Additionally, languages are more related to individual bug categories than bugs overall*.

Functional programming uses mathematical functions and immutability to better structure and reason about software. It is based around the idea that a program is a collection of functions where the output is only dependant on the input (also known as pure function). Which means that we have no internal state that could alter the response to a given input. That is, coding is side effects free. Because of these characteristics code written in this paradigm is normally easier to: reason; test; compose; and, parallelize.

Historically functional programming was mostly exclusive to the academia. However, new strides are being made by communities and some industries to bring it to the mainstream.

Big established languages are integrating some features of this paradigm. Still there is a bigger problem, the knowledge gap that exists in regards to functional programming techniques. Pure functional languages with all their alien patterns are not very attractive to an industry that revolves around object oriented programming and imperative styles of programming. Coding exclusively in these last two styles is tough given that the manipulation of hidden state with no guidelines of control more often than not causes reasoning errors and difficulty in building for the future through concurrency and parallelism. It certainly does not help that most comp sci curriculums do not touch this paradigm and its design patterns. An equal strong multi-paradigm view towards a problem, contrasting the benefits and drawbacks of each, would most definitely result in a better designed system. *Different abstractions for different problems*.

Application programming languages have evolved naturally and gracefully, while system programming languages have not. It is truly unfortunate that the backbone of most of the services we use nowadays is still dominated by languages such as C and C++. These have an enormous baggage of unsafe and legacy constructs. Be it from the standard library, its model of manual memory management, dangling pointers, double free, invalid free, null values deference, lack of a concurrency model, racing conditions, weird edge cases, implicit type conversions, overriding of constness, no module system and lack of a package manager, etc.

By looking at the Common Vulnerabilities and Exposures (CVE) of OpenSSL, Linux, Glibc, Chrome, Internet Explorer, etc we see a pretty dark history of memory corruption (double free, use after free, etc), out of bounds indexing, null pointer dereference, etc. These flaws are very much exploitable and have a huge impact on our daily lives. No matter how knowledgable, it is hard to detect them every single time without the aid of coding tools.

One interesting young systems programming language that tries to bring proven language works to the masses is Rust. It brings much needed features like: memory safety; a typestate system; mutability control; side-effect control; functional mechanisms like high-order functions, closures, pattern matching, lazy iterators, traits, enums which are similar to algebraic data types; ability to opt out of static rules in a clear and precise way; a module system; a better designed standard library from the get go; fast C interoperability; no garbage collector; multi-paradigm; UTF8 strings; zero-cost abstractions; community openness for discussion and contribution, etc.

Rust success is evident on the latest versions of the Firefox Browser. The latter started losing major market share when it couldn't keep up with the coarse-grained parallelism (per tab processes) introduced by Chrome in 2008. This was due to Firefox having an existing code base that needed to be reorganized and restructured to take advantage of multiple cores. This re-architecture feat was deemed Electrolysis and got released to the masses in early-mid 2017. As this was being developed there was ongoing research on how to bring fine-grained parallelism to the browser scene. The problem with separating a program into big tasks is that there might be a load imbalance that causes threads to be idle, thus reducing speed. There is a concern though, smaller tasks have a greater or equal number of synchronization points for a given problem. Thus demanding increased care to the paths that might lead to data races and deadlocks. Rust was the ideal solution since it has threads without data races. So a research project codenamed Servo started around 2012 with the mission to create a modern web engine with fine-

grained isolated tasks written in this language. Servo's great developments and findings culminated into project Quantum, whose purpose is to bring all of these to Firefox through an iterative process. The first Firefox Quantum version was 57 released on 14 of November 2017. It was a huge success: 2x faster than previous Firefox versions and trading blows with Chrome, 30% more memory efficient than Chrome.

Many other interesting projects are being developed by the community and organizations running rust in production (The Friends of Rust). Some of these are: Coursera, Chef, npm, Mozzila, CoreOS, Dropbox, Atlassian, OVH, etc. There is growing enthusiasm and according to the Stack Overflow 2016 and 2017 Results Rust is the most loved language. The future is bright.

From a technical standpoint Rust is a very interesting language since it brings a new way to deal with memory management. Ownership enables Rust to make memory safety guarantees at compile-time without the need of a garbage collector. This is done by the compiler by reasoning on a set of rules and types. Each value has only a single variable at any given time that is called its owner; the assignment of a variable to another moves ownership thus invalidating the previous owner (which means it is not a simple shallow copy), if the element in question is "Copy" (primitive types or any element that implements the Copy trait) it would just be implicitly copied; once the owner's scope ends the element's drop function is called, available by implementing the Drop trait (akin to RAII in C++, avoids valid and suggested use cases of goto in C for complex cleanup, etc). There is no concept of assignment operator overloading or copy constructor which could mean a shallow copy or a deep copy depending on the implementation. To make a deep copy one would have to be explicit and invoke the clone function, available by implementing the Clone trait. By following these rules there is no confusion of who should clean the used resources. It would be extremely convoluted to have to pass an argument to a function and return it back whenever the caller needed this value again, would take two moves. For this reason and others, there is the concept of references and borrowing. References, in a similar way to variables, can be immutable or mutable. By default everything is immutable, and mutation is explicit which is an enormous boost to code comprehension. I know that if a function takes an immutable reference, & (vs &mut), to a struct its fields will not change. References do have rules to guarantee that they are always well behaved: at any given time there can either be one mutable reference or any number of immutable references; and they have to always be valid (lifetime analysis). There is a typical example which is used to elucidate why these reference rules are in place. Imagine you have a vector of ints in C++, and you get a reference to an element in a valid position(& or &mut) and afterwards you push a new value to that vector (&mut self, self represents the struct which gets passed to a struct method implicitly when vec.push(1); similar to the python self keyword but with additional syntax to indicate if it is a immutable or mutable reference, or a simple move). It could happen that the vector was at max capacity and a new chunk of memory got requested to accommodate the new element, so the values that were present in the old heap memory chunk got copied to the new memory and then freed. This results in a dangling pointer since the reference that we saved now points to an invalid memory position. Rust prevents this by guaranteeing that the data will not go away before the references to that data.

There is also a language feature that makes out of bonds indexing easier to prevent, slices. A slice is just a structure with a length and a pointer to a pre-existing collection element. This avoids the use of naked indexes, which go very easily out of sync by applying mutations to the respective collection. One way to deal with this is to return an immutable reference to a slice. This implies that there cannot be a mutation to the collection because there is an immutable reference to it (by the above rules). Out of bounds errors can still happen(&str[0..z], immutable reference to a string slice by applying & and the slice operator to a String, with an invalid z >= len) but they fail fast with a panic at run-time (to avoid exploits, catch errors fast and prevent undefined behaviour) and are more difficult to come across.

Rust enums are another interesting feature of the Rust type system which diverge from the ones found in C and C++. Instead of having enum identifiers (constants) and separate structures to deal with each, Rust has the ability to "bind" them in the same Enum definition. In order words, enums in Rust are identifier + optional struct (an entry in an enum is called a variant in Rust) with the ability to define methods, generic data types, implement traits, etc. As pattern matching is also supported different behaviour can be easily selected depending on the variant. One commonly used enum is Option<T> which holds two variants Some(T) and None. It expresses the idea of presence or absence of value. By levering the type system in this way the compiler can check at compile time if all the cases are handled. This along with the fact that in Rust uninitialized values cannot be used means that errors such as null dereferences which are common in languages that use null, nil, etc are avoided.

Rust also enables robust coding through proper error handling and conventions. It separates errors in two groups recoverable and unrecoverable. Recoverable errors use the Enum Result<T, E> which holds the variants Ok(T) and Err(E). Retrieval of the result implies pattern matching which forces verification of all the possibilities. This is sometimes too verbose so Result contains a set of helper methods that diminish this verbosity, for instance the unwrap method of Result returns T in the case of Ok(T) else calls the panic! macro that stops execution and informs on the error, E. Unrecoverable errors result in a call to panic!, these can happen because of out of bounds indexing, etc. There are also precise guidelines on the Rust docs on how to better handle errors, To panic! or Not to panic!.

Lifetimes are a mechanism that associates scopes to constructs (variables, structs, enums, etc) that hold a reference. This is needed so that the compiler can analyze the validity of references at compile time and prevent dangling pointers,

done by comparing scopes. The compiler's subsystem that does this is called the borrow checker. For instance, if in a nested scope one assigns a reference whose value is defined on that scope to a variable on the outer scope there would be undefined behaviour. Because the outer variable would be referring to a invalid memory position after the inner scope ends, in other words the resource lives less than its references. The compiler in some cases does not determine lifetimes automatically since doing so would increase compile time greatly with little benefit. So lifetime annotations are used. These are needed when there are references on structs or functions. In functions these annotations indicate the relation between the input references' and the output references' lifetimes. In structs it is used in the same way to indicate that a struct reference can not outlive a reference to one of its fields. Lifetime annotations are not always needed for functions because of a set of patterns that are checked by the compiler, the lifetime elision rules.

Rust supports high order functions and closures like some other languages. Because closures capture their environment this can lead to memory safety issues. For this reason there are 3 types of closures which map to the concepts of owner, immutable borrow and mutable borrow. Respectively FnOnce, Fn and FnMut. When a closure is created rust infers the type depending on the use of the values from the environment. If needed the captured values can be forced to move to the closure by adding the keyword move before the closure definition. This is a common pattern when starting a thread.

Iterators are a useful language feature which is also present in Rust. These provide elegant and readable operations on collections, and avoid repetitive collection traversal code through naked indexes. By implementing the Iterator trait many useful methods with default implementations are made available, namely: iterator adapters and consuming adapters. The former generates new iterators, while the latter consumes the iterator elements. Because of the characteristics of ownership there are 3 types of iterators: .iter(), to iterate over immutable references; .iter_mut(), if there is a need for mutation of the references returned by next(); and .into_iter(), to move ownership of the construct into the iterator and return owned values.

In Rust there is also the concept of smart pointers like in C++11. Box<T> works in a similar way to unique_ptr<T>, its job is to ascertain ownership of elements that get allocated in the heap. It is different from the ownership examples seen before because what gets saved on the stack is a pointer to the data allocated on the heap and not the data itself. Boxes allow for recursive types, a type can be recursive by declaring a field with the same type in its definition. This field, however, cannot be allocated on the stack since the compiler wouldn't know how much space to reserve for the associated stack frames. Finite sizes known at compile time are needed. Box<T> works because it is just a wrapper for a pointer, 32 or 64 bits depending on the architecture. The Deref (dereferencing) and Drop (destructor) traits allow for easy use of wrapped pointers. There is also dereferencing coercion, that is, automatic conversion from the smart pointer (Box<T>) to the its generic reference (&T) for easy interoperability between references and smart pointers.

Rc<T> is a reference counted smart pointer, similar to shared_ptr<T>. Its goal is to allow multiple ownership of data. The resource that Rc holds only gets dropped when the inner counter reaches 0. Which means that all the Rc clones (other owners) have been dropped (reached the end of their scope). To guarantee memory safety Rc only provides immutable references. Since reference cycles cause memory leaks (the counter never reaches 0) Weak<T> can be used in a similar way to weak_ptr<T> in C++11 by downgrading one of the owners. The choice between single ownership and multiple immutable references vs shared ownership is a question of understanding the lifetimes of the concerned variables to determine if multiple ownership is needed.

RefCell<T> is a smart pointer that defers borrow checking to the runtime. It makes use of a Rust design pattern called interior mutability to allow mutation to data when there are immutable references to that data, struct.borrow_mut(). Since this breaks some rules (mutation) unsafe code is required. Unsafe code gives more control by turning off certain compiler checks which enforce memory safety. This does not mean that the code is unsafe or memory unsafe, only that the responsibility of code safety is shared between the compiler and the coder. For instance, provide: a safe API (immutability of the outer type for RefCell<T>), data race free code, etc. To conclude, RefCell<T> should only be used when code known to be memory safe fails at compilation (due to the impossibility of analysis of some code properties at compile-time, subset of all memory safe implementations) and compile-time checked constructs (references and some smart pointers) are not appropriate.

Memory and concurrency safeness share some common problems that can be prevented with the ownership and type systems. For instance, having one mutable reference and many immutable references can lead to memory corruption and data races in concurrent or parallel code. Rust's systems guarantee concurrent or parallel programming with no data races, Fearless Concurrency. By default only kernel level threads are supported by the standard library so as not to increase the language runtime unnecessarily and damage interoperability with C. However many other solutions are available in the standard library or in crates: green threads (green), asynchronous programming support through futures (futures-rs), data-parallelism (rayon, similar in a way to OpenMP), message passing (in the standard library), etc. Rust also has Mutex<T> for shared state concurrency synchronization and Arc<T>, atomic rc, for multiple thread ownership of this mutex (Arc is thread safe while rc<T> is not).

Traits are a mechanism similar to Interfaces in Object Oriented Languages that define shared behaviour but with more features like Trait Bounds (used to restrict the generic data types to a subset that implements some traits), Trait Objects (for polymorphism), etc.

REFERENCES

[1] Rust by example. https://rustbyexample.com/. Online.

[2] The rust programming language. https://doc.rust-lang.org/book/second-edition/. Online.

[3] Manish Goregaokar. Fearless concurrency in firefox quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html, Nov 14, 2017. Online.

[4] graydon2. https://graydon2.dreamwidth.org/, Aug. 18th, 2017. Online.

[5] STEVE KLABNIK. Rust is more than safety. http://words.steveklabnik.com/rust-is-more-than-safety, DECEMBER 28, 2016. Online.

[6] Hacker News. Rust creator graydon hoare is now at apple working on swift. https://news.ycombinator.com/item?id=13533701, 2016. Online.

[7] Hacker News. What's a reference in rust? https://news.ycombinator.com/item?id=15802885, 2016. Online.

[8] Hacker News. How firefox got fast again. https://news.ycombinator.com/item?id=15686653, November 2017. Online.

[9] Stack Overflow. Developer survey results 2017. https://insights.stackoverflow.com/survey/2017, 2017. Online.

[10] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. https://cacm.acm.org/magazines/2017/10/221326-a-large-scale-study-of-programming-languages-and-code-quality-in-github/fulltext. Online.

[11] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. https://cacm.acm.org/magazines/2017/10/221326-a-large-scale-study-of-programming-languages-and-code-quality-in-github/fulltext, October 2017. Online.

[12] Project Servo. Technology from the past come to save the future from itself. http://venge.net/graydon/talks/intro-talk-2.pdf, July 2010. Online.