Declan ONUNKWO 05/12/2023

CSC344 – Programming Languages

Assignment: **Memory Management / Perspectives on Rust**

Task 1: The Runtime Stack and the Heap

Rust is an incredibly quick and memory-efficient programming language that can run on embedded devices, power performance-critical applications, and seamlessly interact with other languages. It is a modern systems programming language and is known for its focus on safety, performance, and concurrency. Rust offers a unique blend of low-level control and high-level abstraction. Whether you're a seasoned programmer or new to the language, this conversation will shed light on why Rust has emerged as a compelling choice for systems programming.

Let us start by talking about the runtime stack in Rust. The runtime stack in Rust is in charge of controlling function execution and handling local variables. Every function call generates a new stack frame that holds the function's parameters, local variables, and return address, according to the Last-In-First-Out (LIFO) concept. When a function is called, the stack frames are pushed onto the stack and removed once the function has finished running. By ensuring that references to stack-allocated variables are always valid and correctly handled, Rust's ownership and borrowing system, enforced by the compiler, avoids problems like dangling pointers or use-after-free errors.

Unlike the stack, which handles short-lived data, the heap in Rust is responsible for allocating and managing dynamically-sized and long-lived data. The heap enables the allocation of memory that is accessible from various program components and endures through the end of a

single function. Rust uses ownership, borrowing, and the Box<T> type, which stands for an owned reference to heap-allocated data, to manage heap memory. The ownership system in Rust also maintains memory safety by enforcing stringent guidelines for ownership transfer, avoiding problems like duplicate frees or memory leaks. Additionally, when working with heap-allocated data, Rust offers smart pointers like Rc<T> and Arc<T> for shared ownership, enabling safe concurrency and thread synchronization.

Task 2: Explicit Memory Allocation/Deallocation vs Garbage Collection

Memory management in programming languages can be approached in two main ways: explicit memory allocation/deallocation and garbage collection. Explicit memory management requires developers to manually allocate and deallocate memory, while garbage collection automates the process. In this discussion, we'll explore the differences between these approaches, their implications on programming, and the trade-offs they entail.

Some languages require explicit memory allocation and deallocation. Low-level programming languages such as C and C++ allow you to reserve memory space explicitly for storing data. If you wish to utilize the memory again after you've finished, you must release it. This provides you extra power, but you must be careful not to waste memory. Memory leaks occur when memory is not properly freed. This can lead to problems in the future since the information can be accessed or utilized by other elements of your computer system by accident.

The programmer is completely relieved from the stress of memory management with garbage collection. During runtime, the programming language will handle all the program

requires. As the program runs, memory is allocated and reallocated for any data that is being used, and when that data is no longer being referenced, it is deallocated. While this is highly convenient for the coder, it is a sluggish operation when compared to the above-mentioned explicit memory management. Some languages that have garbage collection include Java and python.

Task 3: Explicit Memory Allocation/Deallocation vs Garbage Collection

The quotes below were directly extracted from https://mmhaskell.com/rust/memory.

- "In Rust, we do allocate memory and de-allocate memory at specific points in our program. Thus, it doesn't have garbage collection, as Haskell does. But it doesn't work quite the same way as C++."
- **"Heap memory always has one owner, and once that owner goes out of scope, the memory gets de-allocated."**
- "When we declare a variable within a block, we cannot access it after the block ends."
- "Another important thing to understand about primitive types is that we can copy them."
- "String literals don't give us a complete string type."
- "We can pass a variable by reference."
- **"If you want a *mutable* reference, you can do this as well. The original variable must be mutable, and then you specify mut in the type signature."**
- **"You can only have a single mutable reference to a variable at a time!"**

- “Deep copies are often much more expensive than the programmer intends. So, a performance-oriented language like Rust avoids using deep copying by default.”
- **“Slices give us an immutable, fixed-size reference to a continuous part of an array…Slices are either primitive data, stored on the stack, or they refer to another object. This means they do not have ownership and thus do not de-allocate memory when they go out of scope.”**

Task 4: Paper Review: Secure PL Adoption and Rust

Rust incorporates aspects from various programming paradigms, including functional, imperative, and object-oriented languages, making it a versatile language with multiple approaches to solve problems. Computer scientists have always been concerned about memory safety and associated issues. To make it easier for lower-level developers to write safe and effective code, languages like Rust and Go were developed by Mozilla to combat memory and safety related vulnerabilities in a simple yet effective manner compared to the complex new languages.

The paper explores Rust's concurrency model, which promotes safe and efficient concurrent programming through the concept of ownership and borrowing. It discusses how Rust's lightweight threads, called "tasks," enable concurrent execution without the overhead of traditional thread-based approaches. Furthermore, it addresses some performance drawbacks of using Rust. It notes that while Rust's performance is generally excellent, certain complex operations may be slower compared to languages like C++. These performance differences are

attributed to the strict safety guarantees provided by Rust, which may require additional runtime checks in some scenarios.

Rust might have a steep learning curve as discussed in this paper, but they were also able to point out ways in which its adoption can be encouraged. One mentioned was that projects should be picked carefully. You do not need to jump into complicated projects without starting small. We also need to be careful and have a good support system. In this modern age, there are many sources of support such as YouTube, stackoverflow, reddit, discord servers and even professors. Lastly, we need to be persistent but patient. People are quick to dismiss things when they do not work for them, but with a little patience and practice, you will be able to reap of your hard work because learning rust is very beneficial in the long run.