Second Problem Set: Memory Management / Perspectives on Rust

Leaning Abstract:

In this problem set, I learned that Rust is like C and C++ because of the stack, heap and similar syntax. It is a low level language and safe. This is because the pointers are taken care of automatically but it doesn't technically have a garbage collection.

Task 1: The Runtime Stack and Heap

The runtime stack and the heap are a big part of programming languages. From prior knowledge, I know that both the stack and heap are used for memory allocation and in order to have good memory management, you have to clear functions on your own. Unlike newer languages like Java, the languages that use stack and heap don't have a garbage collection so you have to manage memory all on your own. Rust is a language that uses the stack and heap. It is safe and allows low level control; something that C and C++ couldn't have both of. Rust is basically like these languages but with newer syntax that guides the programmer towards safety while still allowing low-level control.

A stack is an area in the computer's memory which stores temporary variables created by a function. In the stack, the variables are declared, stored and initialized during runtime. It is a temporary storage memory. When the computing task is complete, the memory of the variable will be automatically erased. Languages like C and C++ use the stack and heap and also use malloc to allocate memory on the heap, however, if you're not careful, it could lead to a lot of problems like memory leaks or dangling pointers. To solve these problems, something called the garbage collector comes into play. It automatically identifies the memory location that is no longer in use and frees up memory. However, this can make the execution time take longer

because the program needs to pause to let the garbage collector do its job. So when Rust executes its program, it creates a stack that keeps track of everything that happens in the program.

A Heap is a memory used by programming languages to store global variables. All global variables are stored in the heap memory space by default. The memory on the heap is dynamically allocated. Dynamic memory allocation is the process of assigning the memory space during the execution time or the run time. Unlike the Stack, the Heap is not managed automatically, it is done by the programmer. An issue with the Heap is that the Heap memory can become fragmented as blocks of memory can become fragmented as blocks of memory are first allocated and then freed. The access speed is slower than the stack, the memory is allocated in a random order, and the access time is slower.

Task 2: “Explicit memory allocation/deallocation vs Garbage Collection”

Explicit memory allocation is, deallocation, the garbage collector. is in many languages like C, C++, Java, and Lisp. Allocation of memory means to allocate memory for data variables to store data. Deallocation of memory is a way to free random access memory of finished processes and allocate new ones. Neither of these will continue to work without the other.

Like many things in programming allocation is taken care of in different ways based on the language but memory allocation can happen either at compile time or for static and global variables or it could be from the Heap. Whenever a new object is being created, memory is allocated in the heap and the pointer is moved to the next memory address. Memory allocated at compile time remains alive until the time process is in execution. A process has to be loaded in the RAM for its execution and remains in the RAM until it's finished. Finished processes are deallocated and new processes are allocated again. In high level, programming deallocation is

done by the garbage collector. A process can not be allocated unless the one before it was deallocated. These two are usually used in low level programming.

The garbage collection(GC) is collection or gaining memory back which has been allocated to objects. It is a process in which programs try to free up memory space that is no longer being used by objects. Garbage collection is implemented differently in every language. Most high level programming languages have some sort of garbage collection built in. While performing garbage collection, all the unwanted objects are destroyed so memory gets freed. The garbage collector takes care of pointing the pointers of freed memory once GC happens. Because the programmer doesn't have to allocate and deallocate space manually, the GC saves time for programmers. However, this can make the execution time take longer because the program needs to pause to let the garbage collector do its job.

Task 3 - Rust : Basic Syntax

1) Rust is a very interesting language to compare to Haskell. It has some similar syntax. But it is not as similar as, say, Elm or Purescript. Rust can also look a great deal like C++. And its similarities with C++ are where a lot of its strongpoints are.

2) We can call a print statement without any mention of the IO monad. We see braces used to delimit the function body, and a semicolon at the end of the statement. If we wanted, we could add more print statements.

3) We'll see that type names are a bit more abbreviated than in other languages. The basic primitives include: Various sizes of integers, signed and unsigned (i32, u8, etc.), Floating

point types f32 and f64 ,Booleans (bool), Characters (char). Note these can represent unicode scalar values (i.e. beyond ASCII) .

4) We mentioned last time how memory matters more in Rust. The main distinction between primitives and other types is that primitives have a fixed size. This means they are always stored on the stack. Other types with variable size must go into heap memory.

5) While variables are statically typed, it is typically unnecessary to state the type of the variable. This is because Rust has type inference, like Haskell!

6) We have type signatures and variable names within the parentheses. Specifying the types of your signatures is **required**. This allows type inference to do its magic on almost everything else. In this example, we no longer need any type signatures in main.

7) Unlike Haskell, it is possible to have an if expression without an else branch.

8) Like Haskell, Rust has simple compound types like tuples and arrays (vs. lists for Haskell). These arrays are more like static arrays in C++ though. This means they have a fixed size. One interesting effect of this is that arrays include their size in their type. Tuples meanwhile have similar type signatures to Haskell:

9) Arrays and tuples composed of primitive types are themselves primitive! This makes sense, because they have a fixed size.

10) Another concept relating to collections is the idea of a slice. This allows us to look at a contiguous portion of an array. Slices use the & operator though.

Task 4 - Rust : Memory Management

1) The suggestion was that Rust allows more control over memory usage, like C++. In C++, we explicitly allocate memory on the heap with new and deallocate it with delete. In Rust, we do allocate memory and deallocate 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++.

2) If you code in Java, C, or C++, this should be familiar. We declare variables within a certain scope, like a for-loop or a function definition. When that block of code ends, the variable is out of scope. We can no longer access it.

3) Rust works the same way(as C). When we declare a variable within a block, we cannot access it after the block ends.

4) Another important thing to understand about primitive types is that we can copy them. Since they have a fixed size, and live on the stack, copying should be inexpensive.

5) But string literals don't give us a complete string type. They have a fixed size. So even if we declare them as mutable, we can't do certain operations like append another string. This would change how much memory they use!

6) Instead, we can use the String type. This is a non-primitive object type that will allocate memory on the heap

7) What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it! We don't need to call delete as we would in C++.

8) Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default.

9) Like in C++, we can pass a variable by reference. We use the ampersand operator (&) for this. It allows another function to "borrow" ownership, rather than "taking" ownership. When it's done, the original reference will still be valid.

10) Slices give us an immutable, fixed-size reference to a continuous part of an array. Often, we can use the string literal type str as a slice of an object String. 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 deallocate memory when they go out of scope.

Task 5 - Rust : Data Types

1) As we've seen so far, Rust combines ideas from both object oriented languages and functional languages. We'll continue to see this trend with how we define data. There will be some ideas we know and love from Haskell. But we'll also see some ideas that come from C++.

2) Rust is a little different in that it uses a few different terms to refer to new data types. These all correspond to particular Haskell structures. The first of these terms is struct.The name struct is a throwback to C and C++. But to start out we can actually think of it as a distinguished product type in Haskell.

3) Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields. The Haskell version would be an "undistinguished product type". This is a type with a single constructor, many fields, but no names.
4) Rust also has the idea of a "unit struct". This is a type that has no data attached to it.

5) The last main way we can create a data type is with an "enum". In Haskell, we typically use this term to refer to a type that has many constructors with no arguments. But in Rust, an enum is the general term for a type with many constructors, no matter how much data each has. Thus it captures the full range of what we can do with data in Haskell.

6) Pattern matching isn't quite as easy as in Haskell. We don't make multiple function definitions with different patterns. Instead, Rust uses the match operator to allow us to sort through these.

7) But unlike Haskell, Rust allows us to attach implementations to structs and enums. These definitions can contain instance methods and other functions. They act like class

definitions from C++ or Python. We start off an implementation section with the impl keyword.

8) We can also create "associated functions" for our structs and enums. These are functions that don't take self as a parameter. They are like static functions in C++, or any function we would write for a type in Haskell.

9) As in Haskell, we can also use generic parameters for our types. Let's compare the Haskell definition of Maybe with the Rust type Option, which does the same thing.

10) Also as in Haskell, we can derive certain traits with one line! The Debug trait works like Show:

Task 6 - Paper Review : Secure PL Adoption and Rust

Languages like Rust were created to combat potentially devastating memory- safety-related vulnerabilities. Such memory vulnerabilities occur in languages like C and C++, while more popular languages like Java enforce memory safety automatically. Mozilla developed Rust to be practical but secure alternatives to C and C++; Rust aims to be fast, low-level, and type- and memory-safe.

Rust can be unsafe. “Since the memory guarantees of Rust can cause it to be conservative and restrictive, Rust provides escape hatches that permit developers to deactivate some, but not all, of the borrow checker and other Rust safety checks.” Unsafe blocks allow the developer to call an unsafe function or method or implement an unsafe trait. Rust is also an incredibly difficult language to learn. Most participants in this article said that Rust was more difficult to learn than other languages. Despite the

high quality of available tools and libraries, Rust still lacks some critical libraries and infras- tructure, perhaps in part because it is fairly new.

Most chose to learn Rust because it is interesting or marketable. Some said they heard about it online but most said that Rust was a useful job skill. The tools are easy to use but they may run a little slow .Also, most survey participants found Rust’s compiler and runtime error messages to be good or very good compared to their reference language. One of the key benefits is that many of Rust’s programmers are confident that the code is safe and correct. It should also be noted that debugging is a huge part of programming when something isn't working. Participants said that they spend a lot less time debugging than in other languages.