18 - Rust

Rust is a multi-paradigm, high-level, general-purpose programming language. Its main focus is on systems programming (providing services to other software), instead of application programming (providing services to the user directly).

It focuses on:

- Performance: no runtime (garbage collector, dynamic typing/binding...), control of low-level details such as memory usage. The performance are comparable to C;
- Safety: type safety, memory safety and especially thread safety;
- Productivity: great tooling, great debugging and error report...;

Rust's syntax is similar to C and C++, although many of its features are more significantly influenced by functional programming languages.

Memory safety

At static time, Rust blocks:

- Null pointers: accessing a variable which does not hold a value. A Null value does not exist in Rust. Data values must always be initialized through a fixed set of forms. It performs checks at compile-time to be sure every branch assign correct values to variables. For nullable types, a generic `Option<T>` type exist, playing the role of Haskell's Maybe or Java's Optional.

```language-rust
enum std::option::Option {
        None,
        Some(T)
}
```

- Dangling pointers: pointers to invalid memory location, resulting in unpredictable results (random results, segmentation fault...). It is address through the concepts of ownership and lifetimes;
- Double frees: a memory location in the heap is reclaimed twice, possibly corrupting the state of the memory manager (leading to crash/modification of execution flow). Rust doesn't allow explicit memory deallocation, and solves it implicitly through the concept of ownership;
- Data races: unpredictable results in concurrent computations;
- Iterator invalidation.

Ownership

Rust uses a stack of activation records, and a heap for dynamically allocated data structures.

To manage the objects in the heap, there is no Garbage Collector: Rust enforces RAII (Resource Acquisition Is Initialization), so whenever an object goes out of scope, its destructor is called and its owned resources are freed.

This is done through the concept of ownership: since variables are in charge of freeing their own resources, resources can only have one owner. This also prevents resources from being freed more than once. Note that not all variables own resources (e.g. references).

But what happens when doing assignments (`let x = y`) or passing function arguments by value (`foo(x)`)? In Rust, this is called a move: the *ownership* of the resources is transferred. After moving resources, the previous owner can no longer be used. This avoids creating dangling pointers. There are exceptions: for primitive types and types

implementing the Copy trait, assignment has a copy semantics, effectively creating a new value, and still making sure that we still have one owner per variable.

Also, by default the variables are immutable (and compiler checks it). Still, it is possible to make variables mutable using the `mut` keyword.

Borrowing

Ownership rules are too restrictive: most of the time, we'd like to access data without taking ownership over it. To accomplish this, Rust uses borrowing: instead of passing objects by value (`T`), objects can be passed by reference (`&T`).

The compiler statically guarantees (via its *borrow checker*) memory safety by allowing only one of the following:

1. Having several immutable references (`&T`) to the object (also known as aliasing).
2. Having one mutable reference (`&mut T`) to the object (also known as mutability).

Regarding the owner:

1. Owner cannot free or mutate its resource while it is immutably borrowed;
2. Owner cannot even read its resource while it is mutably borrowed.

Lifetime

A lifetime is a construct the compiler (via its *borrow checker*) uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is moved/destroyed.

An interesting case are borrowed formal parameters of a function. If borrowed values are returned, each must have a lifetime. The compiled tries to infer lifetimes according to some rules:

1. Borrowed (reference) formal parameters of a function have a lifetime;
2. If there is exactly one input lifetime, it will be assigned to each output lifetime;
3. If a method has more than one input lifetime, but one of them is `&self` or `&mut self`, then this lifetime is assigned to all output lifetimes.

Otherwise, the lifetime must be made explicit using the same syntax of generics. E.g.:

```language-rust
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
        if s1.len() > s2.len() {
                s1
        } else {
                s2
        }
}
```

Traits

A `trait` is a collection of methods defined for an unknown type: `Self`. They are equivalent to Type Classes in Haskell and to Concepts in C++20, similar to Interfaces in Java.

A trait can include abstract and concrete (default) methods. It cannot contain fields/variables. It can access other methods declared in the same trait.

A struct can implement a trait providing an implementation for at least its abstract methods:

```language-rust
impl <TraitName> for <StructName>{ … }
```

The compiler is capable of providing basic implementations for some traits via the `#[derive]` attribute. These traits can still be manually implemented if a more complex behaviour is required. Some examples are: `Eq, Ord, Clone, Copy, Debug`.

Note that in Rust, a trait must be implemented separately for each type that needs to use it, while in Haskell, a type can automatically be an instance of a type class if it satisfies the constraints of the type class.

Bounded polymorphism with traits

Example:

```language-rust
trait Stack<T> {
        fn new() -> Self;
        fn is_empty(&self) -> bool;
        fn push(&mut self, data: Box<T>);
        fn pop(&mut self) -> Option<Box<T>>;
}

// Traits are used here:
fn generic_push<T, S: Stack<T>>(stk: &mut S,
        data: Box<T>) {
        stk.push(data);
}

fn main() {
        let mut stk = SLStack::<u32>::new();
        let data = Box::new(2048);
        generic_push(&mut stk, data);
}
```

Rust uses monomorphization: a separate copy of the code is generated for each instantiation of a function. This is in contrast with the type erasure of Java.

- Pros: optimized code for each specific use case;
- Cons: increased compile time and size of the resulting binaries.

Closures

Closures are anonymous functions that can capture the enclosing environment.

```language-rust
let x = 42;

// A regular function can't refer to variables in the enclosing environment
//fn function(i: i32) -> i32 { i + outer_var }
// TODO: uncomment the line above and see the compiler error. The compiler
// suggests that we define a closure instead.

// Closures are anonymous, here we are binding them to references
// Annotation is identical to function annotation but is optional
// as are the `{}` wrapping the body. These nameless functions
// are assigned to appropriately named variables.
let closure_annotated = |i: i32| -> i32 { i + outer_var };
```

Smart Pointers

Smart pointers are data structures that act like a pointer but also have additional metadata and capabilities.

Box

All values in Rust are stack allocated by default. Values can be boxed (allocated on the heap) by creating a `Box<T>`. A box is a smart pointer to a heap allocated value of type `T`. When a box goes out of scope, its destructor is called, the inner object is destroyed, and the memory on the heap is freed. Boxed values can be dereferenced using the `*` operator, which returns the value.

Boxes are useful when:

- Size of data is not known statically (e.g. recursive types);
- We have big data, and we want to perform `move` rather than `copy`.

Rc

There are cases when a single value might have multiple owners. For example, in graph data structures, multiple edges might point to the same node, and that node is conceptually owned by all of the edges that point to it.

When multiple ownership is needed, Rc (Reference Counting) can be used. `Rc` keeps track of the number of the references which means the number of owners of the value wrapped inside an `Rc`.

Reference count of an `Rc` increases by 1 whenever an `Rc` is cloned, and decreases by 1 whenever one cloned `Rc` is dropped out of the scope. When an `Rc`'s reference count becomes zero (which means there are no remaining owners), both the `Rc` and the value are all dropped.

Note that cloning an `Rc` never performs a deep copy. Cloning creates just another pointer to the wrapped value, and increments the count.

Arc

When shared ownership between threads is needed, Arc (Atomically Reference Counted) can be used. This struct, via the `Clone` implementation can create a reference pointer for the location of a value in the memory heap while increasing the reference counter. As it shares ownership between threads, when the last reference pointer to a value is out of scope, the variable is dropped.

To implement mutability, `Arc` implements the 2 following traits:

- Send: safe to send it to another thread.
- Sync: safe to share between threads (T is Sync if and only if `&T` is Send).

However, race conditions could still occur. In that care, the classic solution is to make use of Mutex (a mutual exclusion primitive useful for protecting shared data) wrapped in an `Arc`.

RefCell

The `RefCell<T>` type represents a mutable memory location with dynamically checked borrow rules: compared to references/boxes, where the borrowing rules are checked at compile time, with `RefCell` they are enforced at run time.

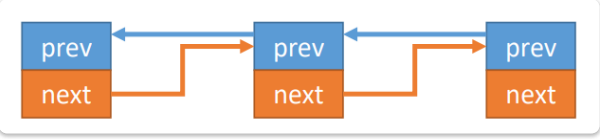
In order to do it, Rust make use of the Interior mutability design pattern, which allows to mutate data even when there are immutable references to that data. Normally, this action is disallowed by the borrowing rules. To mutate data, the pattern uses unsafe code inside a data structure to bend Rust's usual rules that govern mutation and borrowing.

`RefCells` has methods `borrow()` and `borrow_mut()` which return a smart pointer (`Ref` or `RefMut`). `RefCell` keeps track of how many `Ref` and `RefMut` are active, and panics (crashes) if the ownership/borrowing rules are invalidated.

A common way to use `RefCell` is in combination with `Rc`. Recall that Rc lets you have multiple owners of some data, but it only gives immutable access to that data. If you have an Rc that holds a RefCell, you can get a value that can have multiple owners and that you can mutate!

Raw pointers

They are used when mutably sharing is required. For example, with doubly linked lists:




```language-rust
struct Node {
        prev: option<Box<Node>>,
        next: *mut Node
}
```

In order to make use of them though, code should be encapsuled in an unsafe block.