Lenguaje Rust - Materia Teoría del Lenguaje

1. Origen

Rust es un lenguaje de programación que fue desarrollado por Mozilla Research, con su primera versión estable (1.0) lanzada en mayo de 2015.

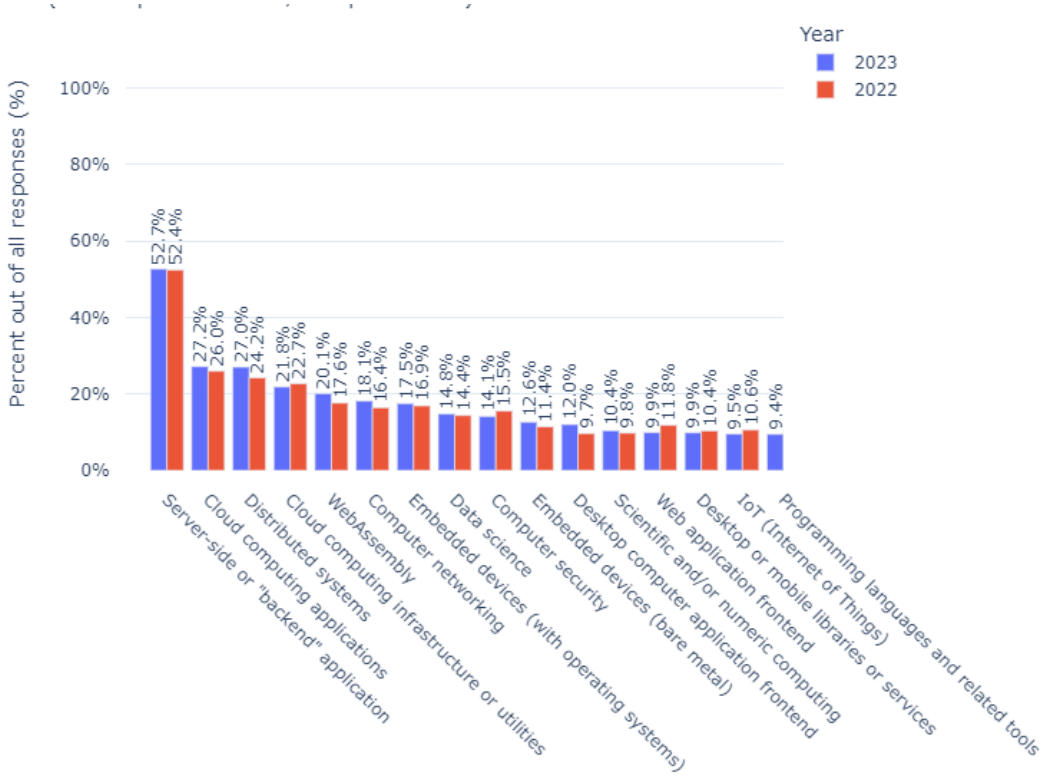
La motivación detrás de la creación de Rust fue principalmente resolver problemas de seguridad y gestión de memoria que son comunes en lenguajes como C y C++, sin sacrificar el rendimiento. Rust se diseñó para ser un lenguaje seguro, concurrente y práctico, enfocado en ofrecer una experiencia óptima tanto para sistemas embebidos como para aplicaciones de alto rendimiento.

2. Para qué sirve (y para qué no)

Bueno para:

- Sistemas Embebidos y Programación de Sistemas.
- Aplicaciones Concurrentes.
- Aplicaciones de Alto Rendimiento.

Además, según la encuesta anual de Rust, los usuarios han determinado que Rust es popular para crear server backends, servicios de web y networking y tecnologías cloud.



Malo en:

- Desarrollo Rápido de Prototipos: Debido a su enfoque en la seguridad y la gestión precisa de recursos, Rust puede ser más complejo y menos flexible para el desarrollo rápido en comparación con lenguajes interpretados como Python o scripts rápidos.
- Aplicaciones con Alta Interfaz Gráfica de Usuario (GUI): Aunque hay esfuerzos en este sentido, las bibliotecas y herramientas para desarrollo de GUIs en Rust aún no son tan maduras como las de otros lenguajes como JavaScript (con Electron), Swift, o Kotlin.

3. Características básicas del Lenguaje

3.1. Paradigmas que soporta

El lenguaje Rust soporta programación imperativa, funcional y concurrente.

Ejemplo de programación imperativa en nuestro código:

```rust
/// Funcion que se encarga de crear un usuario
async fn create_user(info: web::Json<(String, String)>, sistema: web::Data<Arc<Sistema>>) -> impl Responder {
    let (email: String, password: String )= info.into_inner();
    if sistema.user_already_exists(&email) {
        return HttpResponse::Conflict().body(format!("User with email {} already exists", email));
    }

    sistema.add_client(email.clone(), password.clone());

    HttpResponse::Ok().body(format!("User created"))
}
```

En la función create_user, se realizan acciones secuenciales para verificar si un usuario existe. En caso de que no exista, se agrega, cambiando el estado del sistema.

Ejemplo de programación funcional en nuestro código:

```rust
/// Funcion que se encarga de listar todas las habitaciones
async fn list_all_rooms(sistema: web::Data<Arc<Sistema>>) -> impl Responder {
    let rooms: Vec<Habitacion> = sistema.get_all_rooms();
    print!("ListAllRooms => {:?}", rooms);
    HttpResponse::Ok().json(rooms)
}
```

En la función list_all_rooms, utilizamos la función get_all_rooms, que devuelve un valor basado en su entrada, sin efectos secundarios.

Ejemplo de programación concurrente en nuestro código:

```rust
// Spawn a task to listen for stop signal
tokio::spawn(async move {
    rx.clone().changed().await.unwrap();
    // Signal received, time to shut down
    server_stop_flag_clone.store(true, Ordering::Relaxed);
    if let Err(err: Error) = sistema.save_reservations_to_csv("reservas.csv") {
        eprintln!("Error saving reservations: {}", err);
    }
});
```

En la función main de nuestro server, utilizamos tokio::spawn, para crear una tarea asincrónica que se ejecuta concurrentemente.

En cuanto a la comparativa con otros lenguajes, lenguajes como C/C++ soportan los paradigmas imperativo, orientado a objetos (en C++), funcional limitado y concurrente. Oz, por su parte, soporta una combinación de paradigmas imperativo, declarativo (funcional y lógico) y concurrente.

Ejemplo de programación concurrente en estos lenguajes puede ser

En C/C++

```
std::thread t([](){
    while (running) {
        // Do something concurrently
    }
});
t.join();
```

En Oz

```
thread
   {Wait Changed}
   ServerStopFlag := true
end
```

3.2. Compilado/ Interpretado

El lenguaje Rust usa “rustc” para compilar el código.

```
rustc main.rs
./main
```

El lenguaje C usa “gcc”, similar a C++ que utiliza gcc++.

```
gcc main.c -o main
./main
```

Se diferencian de Oz, que es interpretado, usando el sistema Mozart.

```
ozc -c main.oz
ozengine main.ozf
```

3.3. Tipado

En rust el tipado es estático y fuerte. Al declarar variables, puede o no explicitarse el tipo, y en caso de que esa variable se modifique a lo largo del programa, se declara con un “mut” adelante.

```
let _email_validator: EmailValidator = EmailValidator;
let _password_validator: PasswordValidator = PasswordValidator;
let mut email: String = String::new();
let mut password: String = String::new();
```

C y C++ tienen también tipado estático y fuerte. Oz, por su parte, tiene tipado dinámico y fuerte.

```
declare
X = 10
{Browse X}
```

3.4. Control de Flujos

Rust soporta if, if let, else, match, loop, while y for.

Ejemplo de match (y loop) en nuestro código:

```rust
pub fn ask_dates_and_number_guest() -> (String, String, u8) {
    let mut date_start: String = String::new();
    let mut date_end: String = String::new();
    let mut cant_integrantes: String = String::new();

    let DateValidator = DateValidator;
    print!("Enter start date (YYYY-MM-DD): ");
    loop {
        io::stdout().flush();
        io::stdin().read_line(buf: &mut date_start);
        match DateValidator.validate(input: &date_start) {
            Ok(_) => break,
            Err(e: ValidationError) => println!("{}", e),
        }
        date_start.clear();
    }
    print!("Enter end date (YYYY-MM-DD): ");
    loop {
        io::stdout().flush();
        io::stdin().read_line(buf: &mut date_end);
        match DateValidator.validate(input: &date_end) {
            Ok(_) => break,
            Err(e: ValidationError) => println!("{}", e),
        }
        date_end.clear();
    }
    print!("Enter number of guests: ");
    loop {
        io::stdout().flush();
        io::stdin().read_line(buf: &mut cant_integrantes);
        match cant_integrantes.trim().parse::<u8>() {
            Ok(_) => break,
            Err(_) => println!("Invalid number of guests. Please enter a valid number"),
        }
        cant_integrantes.clear();
    }
    let ucant_integrantes: u8 = cant_integrantes.trim().parse::<u8>().unwrap();

    (date_start.trim().to_owned(), date_end.trim().to_owned(), ucant_integrantes)

} fn ask_dates_and_number_guest
```

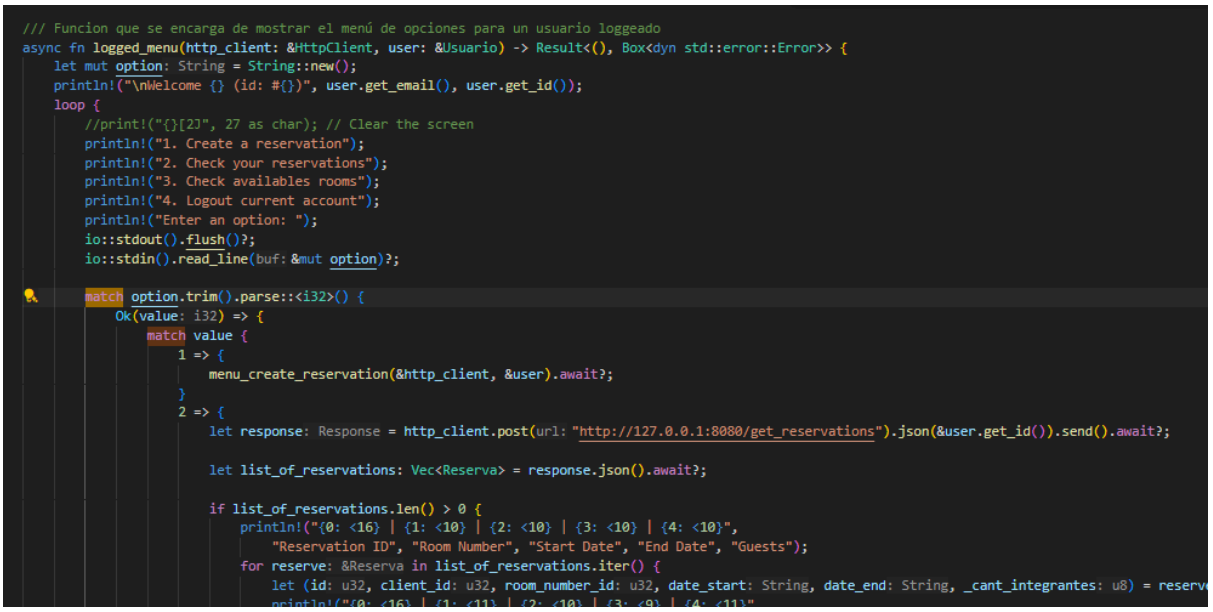
En este caso, la función ask_dates_and_number_guest solicita al usuario que ingrese una fecha de inicio, una fecha de fin y el número de huéspedes. La función valida los datos ingresados, y en caso de que los datos sean inválidos, solicita al usuario que los ingrese nuevamente.

match es una construcción de control de flujo que compara el resultado de la condición o variable que se le pase, con diferentes patrones. Tiene dos resultados, en nuestro caso:

- Si el resultado es Ok(_), el bucle se interrumpe con break.
- Si el resultado es Err(e), imprime el error e y el bucle continúa, solicitando la fecha nuevamente.

loop es un bucle infinito que solo se detiene si se encuentra un break. En el código, se utiliza para solicitar y validar repetidamente la entrada del usuario hasta que se proporcione una entrada válida.

Ejemplo de pattern matching (en el video lo mostramos bien entero)



```rust
/// Funcion que se encarga de mostrar el menú de opciones para un usuario loggeado
async fn logged_menu(http_client: &HttpClient, user: &Usuario) -> Result<(), Box<dyn std::error::Error>> {
    let mut option: String = String::new();
    println!("\nWelcome {} (id: #{})", user.get_email(), user.get_id());
    loop {
        //print!("{}[2J", 27 as char); // Clear the screen
        println!("1. Create a reservation");
        println!("2. Check your reservations");
        println!("3. Check availables rooms");
        println!("4. Logout current account");
        println!("Enter an option: ");
        io::stdout().flush()?;
        io::stdin().read_line(buf: &mut option)?;

        match option.trim().parse::<i32>() {
            Ok(value: i32) => {
                match value {
                    1 => {
                        menu_create_reservation(&http_client, &user).await?;
                    }
                    2 => {
                        let response: Response = http_client.post(url: "http://127.0.0.1:8080/get_reservations").json(&user.get_id()).send().await?;

                        let list_of_reservations: Vec<Reserva> = response.json().await?;

                        if list_of_reservations.len() > 0 {
                            println!("{0: <16} | {1: <10} | {2: <10} | {3: <10} | {4: <10}",
                                "Reservation ID", "Room Number", "Start Date", "End Date", "Guests");
                            for reserve: &Reserva in list_of_reservations.iter() {
                                let (id: u32, client_id: u32, room_number_id: u32, date_start: String, date_end: String, _cant_integrantes: u8) = reserv
```

Los lenguajes C y C++ soportan if, else, switch, while, for, y do-while. Oz soporta if, case, for y while.

3.5. TDAs

Rust se maneja con structs y enums.

```rust
struct ReservationRequest {
    client_id: u32,
    room_number: u32,
    date_start: String,
    date_end: String,
    cant_integrantes: u8,
}
```

```rust
pub enum AppErrors {
    ServerConnection,
    ClientConnection,
    LostConnection,
    UnidentifiedPacket,
    DeserializingPacket,
    AuthenticationError,
    PublishingError,
    SubscribingError,
    PacketSendingError,
    PingError,
    None,
}
```

C se maneja con structs, C++ con clases y enums. Oz utiliza registros (record).

3.6. Manejo de memoria

En Rust existen dos conceptos fundamentales que gestionan cómo se usa la memoria en el programa: Ownership y Borrowing.

Ownership se refiere a la propiedad exclusiva de un valor. Solo un propietario puede modificar o liberar el recurso al que apunta. Cuando se transfiere la propiedad de una variable a otra, se dice que la variable anterior ha sido "movida" y ya no se puede usar.

```rust
/// Funcion principal que se encarga de iniciar el servidor
#[actix_web::main]
▶ Run | Debug
pub async fn main() -> std::io::Result<()> {
    let sistema: Arc<Sistema> = Arc::new(Sistema::new());
```

Acá en la función main del servidor, la variable sistema tiene la propiedad del nuevo objeto Sistema, envuelto en un Arc.

=> Arc (Atomic Reference Counting) es un contenedor que permite que múltiples threads compartan la propiedad de un valor. Utiliza contadores de referencia atomicos para gestionar el acceso concurrente.

Luego al hacer sistema_clone.clone(), se transfiere una copia del Arc<Sistema> a la nueva aplicación Atix. (tx.clone() hace lo mismo para el canal de comunicación).

```rust
let servidor: Server = HttpServer::new(move || {
    App::new()
        .app_data(web::Data::new(sistema_clone.clone()))
        .app_data(web::Data::new(tx.clone()))
        .route("/create_user", web::post().to(create_user)) // Nueva ruta para crear usuario
        .route("/login", web::post().to(login_user))
        .route("/list_all_rooms", web::post().to(list_all_rooms))
        .route("/get_reservations", web::post().to(get_reservations))
        .route("/check", web::post().to(check_availability)) // Nueva ruta para verificar disponibilidad
        .route("/reserve", web::post().to(create_reservation)) // Nueva ruta para crear reserva
        .route("/exit", web::get().to(stop_server)) // Manejar solicitud especial
})
.bind("127.0.0.1:8080")? HttpServer<impl Fn() -> App<…>, …, …, …>
.run();
```

Borrowing permite que varias referencias lean o una referencia modifique un valor sin tomar la propiedad. Rust asegura que mientras haya una referencia modificable, no puede haber referencias de solo lectura y viceversa.

- Referencias inmutables en funciones

```rust
/// Funcion que se encarga de crear un usuario
async fn create_user(info: web::Json<(String, String)>, sistema: web::Data<Arc<Sistema>>) -> impl Responder {
    let (email: String, password: String )= info.into_inner();
    if sistema.user_already_exists(&email) {
        return HttpResponse::Conflict().body(format!("User with email {} already exists", email));
    }

    sistema.add_client(email.clone(), password.clone());

    HttpResponse::Ok().body(format!("User created"))
}
```

En la función create_user, sistema: web::Data<Arc<Sistema>> es una referencia compartida e inmutable a Arc<Sistema>. Se permite que múltiples referencias lean el Sistema.

- Referencias inmutables dentro del Closure

```rust
let servidor: Server = HttpServer::new(move || {
    App::new()
        .app_data(web::Data::new(sistema_clone.clone()))
        .app_data(web::Data::new(tx.clone()))
        .route("/create_user", web::post().to(create_user)) // Nueva ruta para crear usuario
        .route("/login", web::post().to(login_user))
        .route("/list_all_rooms", web::post().to(list_all_rooms))
        .route("/get_reservations", web::post().to(get_reservations))
        .route("/check", web::post().to(check_availability)) // Nueva ruta para verificar disponibilidad
        .route("/reserve", web::post().to(create_reservation)) // Nueva ruta para crear reserva
        .route("/exit", web::get().to(stop_server)) // Manejar solicitud especial
})
.bind("127.0.0.1:8080")? HttpServer<impl Fn() -> App<…>, …, …, …>
.run();
```

Volviendo a este ejemplo, sistema_clone es una referencia inmutable a Arc<Sistema> que se clona y se pasa al nuevo App.

4. Características avanzadas del lenguaje

4.1. Manejo de Concurrencia

Rust utiliza “Arc” y “Mutex” para evitar errores en la programación concurrente. Arc permite compartir datos entre múltiples threads de forma segura. Mutex (Mutual Exclusion) permite proteger datos compartidos para garantizar que solo un thread acceda a los datos a la vez.

```rust
pub struct Sistema {
    reservations: Mutex<Vec<Reserva>>,
    clients: Mutex<HashMap<u32, Usuario>>,
    rooms: Mutex<Vec<Habitacion>>,
    next_client_id: Mutex<u32>,
    next_reservation_id: Mutex<u32>,
    files_and_headers: Vec<(String, Vec<&'static str>)>,
}
```

```rust
/// Guarda las reservas en un archivo CSV.
pub fn save_reservations_to_csv(&self, filename: &str) -> Result<(), io::Error> {
    let reservations: MutexGuard<Vec<Reserva>> = self.reservations.lock().unwrap();
```

self.reservations.lock().unwrap(): como reservations es un Mutex, lock().unwrap() intenta bloquear el mutex y obtener una referencia mutable a las reservas. Si falla al bloquear el mutex (por ejemplo, si otro thread ya lo tiene bloqueado y ha ocurrido un deadlock), unwrap() provocará que el programa falle con un mensaje de error.

4.1.1. Uso de Threads => *Agregado del TP para mostrar en el video*

Un thread es una secuencia de instrucciones que puede ser ejecutada de manera independiente dentro de un proceso. Después de crear un thread, se obtiene un JoinHandle que permite esperar a que el thread termine su ejecución utilizando el método join.

Como no implementamos threads en el trabajo, la idea es mostrar en el video una simulación de operación de carga intensiva de datos en un thread aparte mientras el servidor web sigue respondiendo a las solicitudes HTTP. (Aca explicaría que se tiene q importar la biblioteca, hacer spawn, etc)

4.1.2. Uso de async

Permite la programación asíncrona y concurrente de manera segura y eficiente. La programación asíncrona es crucial para construir aplicaciones que pueden manejar muchas operaciones de entrada/salida (I/O) concurrentes sin bloquear el hilo de ejecución principal.

Las funciones create_user, login_user, list_all_rooms, get_reservations, check_availability y create_reservation están marcadas con async fn. Esto indica que cada una de estas funciones puede suspenderse en puntos de espera (await) sin bloquear el hilo principal mientras espera que se completen las distintas operaciones.

El servidor HttpServer de Actix-web es asíncrono, lo que significa que puede manejar múltiples solicitudes simultáneamente sin bloquear. Esto se logra al usar tokio::spawn para ejecutar tareas concurrentemente, como escuchar señales de detención (rx.clone().changed().await) o esperar que el servidor se detenga (servidor.await).

```rust
// Spawn a task to listen for stop signal
tokio::spawn(async move {
    rx.clone().changed().await.unwrap();
    // Signal received, time to shut down
    server_stop_flag_clone.store(true, Ordering::Relaxed);
    if let Err(err: Error) = sistema.save_reservations_to_csv("reservas.csv") {
        eprintln!("Error saving reservations: {}", err);
    }
});

let _ = tokio::spawn(async move {
    if server_stop_flag.load(Ordering::Relaxed) {
        return Err(std::io::Error::new(std::io::ErrorKind::Interrupted, "Server stopping signal received".to_string()));
    }
    servidor.await
})
.await;
```

4.2. Errores

Rust utiliza dos tipos principales de manejo de errores: Result y Option. Result se utiliza para operaciones que pueden fallar. Tiene las variantes Ok(T) y Err(e). Option se usa para valores que pueden estar o no presentes, sus variantes son Some(T) y None.

```rust
/// Guarda las reservas en un archivo CSV.
pub fn save_reservations_to_csv(&self, filename: &str) -> Result<(), io::Error> {
    let reservations: MutexGuard<Vec<Reserva>> = self.reservations.lock().unwrap();
    let file: File = File::create(path: filename)?;
    let writer: BufWriter<File> = BufWriter::new(inner: file);
    let mut csv_writer: Writer<BufWriter<File>> = WriterBuilder::new().from_writer(wtr: writer);

    for reservation: &Reserva in reservations.iter() {
        csv_writer.serialize(record: reservation)?;
    }

    csv_writer.flush()?;
    Ok(())
}
```

Result<(), io::Error>: La función devuelve un Result donde el tipo de éxito (Ok) es () (un tipo vacío que indica éxito sin un valor de retorno) y el tipo de error (Err) es io::Error. Esto significa que la función puede devolver un error de tipo io::Error si algo sale mal al intentar guardar las reservas en el archivo CSV.

4.3. Conceptos Avanzados no incluidos en nuestro trabajo

Lifetime annotations: Controlan cuánto tiempo vive una referencia para garantizar que no existan referencias colgantes.

```rust
// Función que devuelve la referencia más larga entre dos strings
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

▶ Run | Debug
fn main() {
    let string1: String = String::from("Rust");
    let string2: String = "Es genial".to_string();

    let result: &str = longest(x: &string1, y: &string2);

    println!("La cadena más larga es: {}", result);
}
```

Generics: Permiten escribir funciones y tipos que pueden trabajar con cualquier tipo de datos.

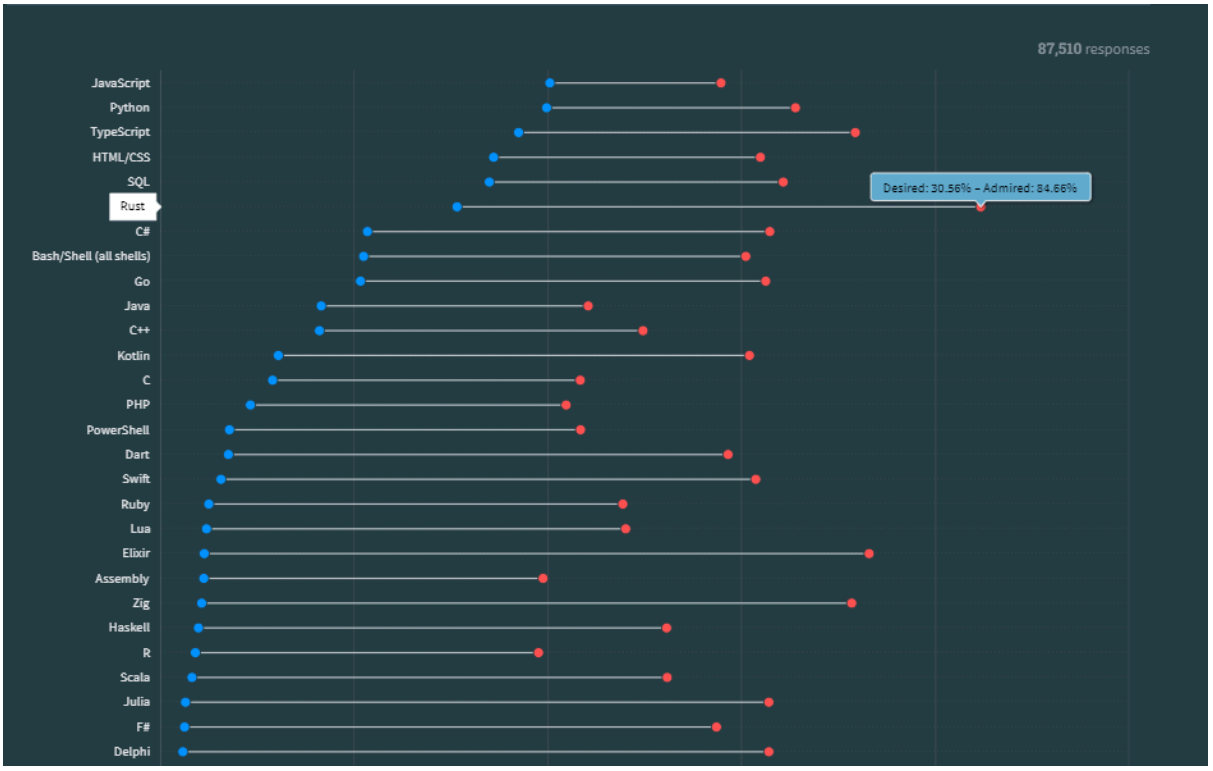
```rust
// Función genérica para devolver el elemento más grande en una lista
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest: &T = &list[0];
    for item: &T in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

▶ Run | Debug
fn main() {
    let numbers: Vec<i32> = vec![1, 3, 5, 2, 4];
    let result: &i32 = largest(list: &numbers);
    println!("El número más grande es: {}", result);

    let chars: Vec<char> = vec!['a', 'c', 'f', 'b', 'e'];
    let result: &char = largest(list: &chars);
    println!("La letra más grande es: {}", result);
}
```

Traits: Proporcionan una forma de definir comportamiento compartido entre diferentes tipos. Serían como las Interfaces en C++

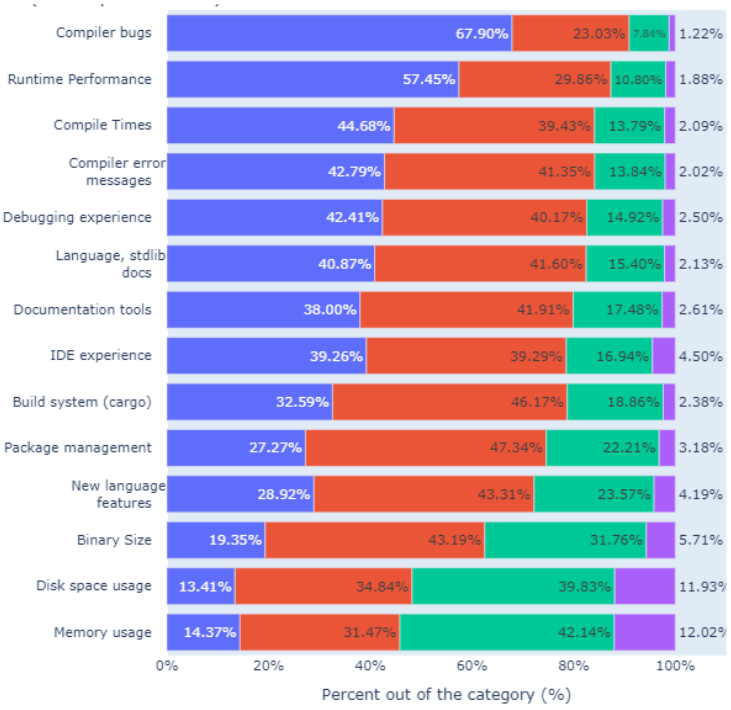
```rust
// Definición de un trait `Animal` con un método `sound`
1 implementation
trait Animal {
    fn sound(&self) -> &'static str;
}

// Implementación del trait para el tipo `Dog`
1 implementation
struct Dog;

impl Animal for Dog {
    fn sound(&self) -> &'static str {
        "Woof!"
    }
}

▶ Run | Debug
fn main() {
    let dog: Dog = Dog;
    println!("El perro hace: {}", dog.sound());
}
```

5. Estadísticas

De acuerdo al Stack Overflow Survey del año 2023, Rust se posiciona en el lenguaje más “admirado”, sosteniendo que más del 80% de los usuarios volverían a usarlo.



Según las encuestas realizadas por Rust, los usuarios sostienen que el tiempo de compilación debe mejorarse.



En cuanto a la performance de Rust en comparación con otros lenguajes, encontramos un ejemplo de un usuario que compara la performance de Rust, C# y Go, titulado “ Performance Comparison of Rust, C#, and Go for High-Performance Web APIs in Kubernetes Cluster”. Como indica su nombre, el ejemplo está basado en la creación de una API web de alto rendimiento en el clúster de Kubernetes.

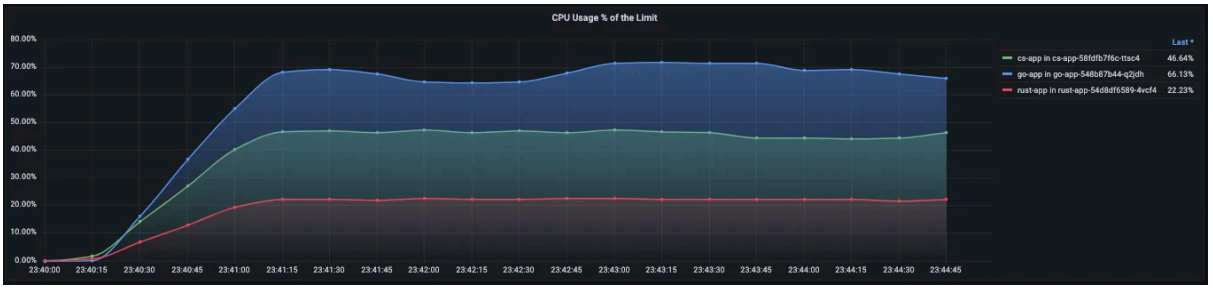
Para la creación de las APIs, para Rust y C# se utilizó el marco Actix y el marco ASP.NET Core respectivamente. Ambos tienen métodos de almacenamiento y recuperación.

Además, Rust usa el marco Actix y la biblioteca Serde, mientras que C# usa el marco ASP.NET Core y Entity Framework Core para la interacción con la base de datos.

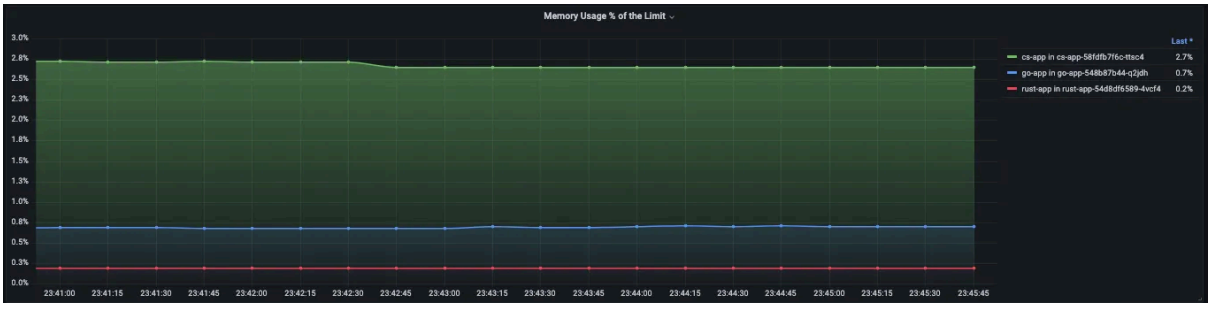
Las observaciones muestran diferencias en el uso de CPU y memoria entre Rust, C# y Go tanto para el estado de carga.

Los lenguajes son representados por el color azul (Go), verde (C#) y rojo (Rust). Las observaciones indican:

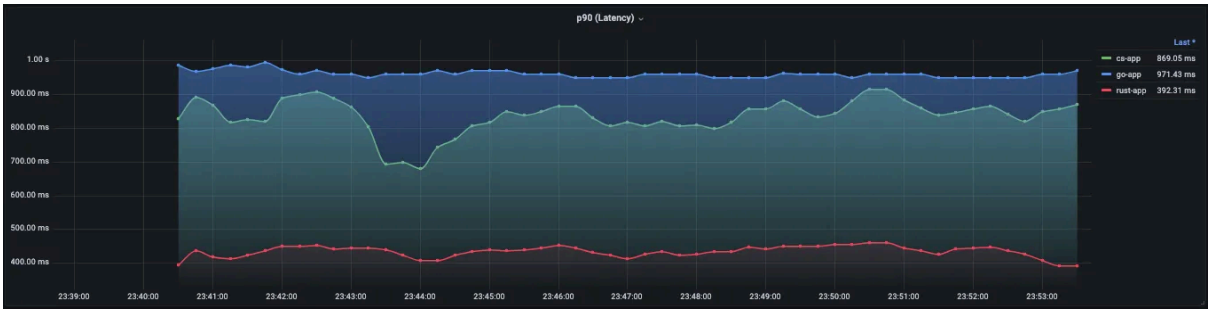
Go es el lenguaje que tiene más uso de CPU post load.



C# es el lenguaje que tiene más uso de memoria post load.



Rust es el lenguaje con menor latencia.



Se concluye en este ejemplo que Rust demuestra consistentemente un mejor rendimiento que C# y Go en los puntos de referencia.

6. Casos de Estudio

Dropbox

Dropbox

Varios componentes del sistema principal de almacenamiento de ficheros de Dropbox están escritos en Rust, como parte de un proyecto más amplio para aumentar la eficiencia de sus centros de datos. Actualmente es usado por todo el almacenamiento de Dropbox, que sirve a más de 500 millones de usuarios.

Yelp

yelp

Yelp ha desarrollado un *framework* en Rust para tests A/B en tiempo real. Se usa en todos las páginas y apps de Yelp, para lanzar experimentos en áreas desde UX hasta infraestructura interna. Eligieron Rust porque es tan rápido como C (bajo coste de ejecución) y más seguro que este (bajo coste de mantenimiento).