Análisis estático extendido de código Rust

Ezequiel Giachero

23 de marzo de 2023

Índice general

Capítulo 1

Introducción

1.1. El lenguaje de programación Rust

Rust es un lenguaje de programación moderno, compilado, multiparadigma y de propósito general. Este comenzó siendo desarrollado por Mozilla, y vio sus primeras versiones hace más de una década en el 2010, siendo sus mayores influencias C++, Haskell y Erlang. Este lenguaje ha tenido un gran impacto en la comunidad de programadores, y ha sido votado como el "lenguaje de programación mas querido" en Stack Overflow desde el año 2016 hasta el 2022.

Rust fue diseñado desde el comienzo teniendo una visión muy enfocada en el rendimiento, la concurrencia y especialmente en la seguridad. Tan es así que implementa características únicas, como lo es su sistema de tipos basado en *ownership*, junto a la herramienta del *borrow-checker*, que permite a Rust garantizar la seguridad de la memoria y de los hilos de ejecución y eliminar muchos fallos en tiempo de compilación, además de ser muy rápido y eficiente porque evita los problemas que trae el uso del garbage-collector utilizado en muchos lenguajes modernos. Gracias a esto y a sus mecanismos de manejo de memoria de bajo nivel, hoy en día Rust es considerado como una alternativa (inclusive más segura) a C y C++. Por esto grandes empresas como Microsoft, Amazon y Google utilizan Rust en muchos de sus proyectos. Al ser multi-propósito se utiliza en muchos ámbitos, pero especialmente es utilizado en la creación de programas para sistemas embebidos y servidores webs.

Sin embargo, para mantener la seguridad en todos los programas escritos en Rust, el sistema de tipos de Rust viene asociado a con muchas restricciones y comprobaciones. Estas si bien traen muchas ventajas y forman parte de las bases de Rust, hacen que resulte muy complejo o incluso imposible crear ciertas estructuras o funciones respetándolas. Es por esto que surge la necesidad del código unsafe en Rust, donde se deshabilitan varios de los chequeos del *borrow-checker* y queda en responsabilidad del programador crear programas seguros y sin fallos.

"Rust puede ser pensado como la combinación de dos lenguajes de programación: Safe Rust y Unsafe Rust. Convenientemente, estos nombres significan exactamente lo que dicen: Safe Rust es seguro. Y Unsafe Rust, no lo es ... Safe Rust es el *verdadero* lenguaje y Unsafe Rust es exactamente como Safe Rust con las mismas reglas y semánticas, solo que te permite hacer unas cosas extra que de definitivamente no son seguras" (Beingessner and Klabnik, 2016)

En un lenguaje donde la seguridad imperativa, dejarla en mano de los programadores no es la solución ideal ya que se pueden cometer errores difíciles de detectar y que pongan en riesgo la estabilidad del sistema en general. Este problema ha sido detectado y está siendo atacado por la comunidad de Rust, mediante ciertos mecanismos y en proyectos como **MIRI** o **Rudra**, que contienen una gran cantidad de herramientas para detectar errores específicos en código unsafe. Estas herramientas, sin embargo, toman un enfoque más dinámico y dependen de otros factores, como por ejemplo la disponibilidad de tests para poder realizar los análisis necesarios.

El presente trabajo tiene como objetivo crear una herramienta para el análisis estático del código Rust; que permita analizar bloques de un programa que contengan código unsafe y su contexto inmediato, para encontrar errores o fallos de los cuales, si los mecanismos de seguridad no hubieran sido deshabilitados, deberían haber sido captados y reportados por el compilador. En especial, nos enfocamos en encontrar problemas de aliasing utilizando el método points-to, mediante Stacked Borrows problemas de referencias colgadas que el borrow-checker se habría encargado y verificar si se producen transformaciones de variables inseguras que puedan producir comportamientos inesperados.

1.2. Fundamentos de Rust

1.2.1. Move semantics

Ownership es un conjunto de reglas que dictan como el lenguaje de Rust maneja la memoria. Como mencionamos en anteriormente es una alternativa a los garbage collector que utilizan muchos otros lenguajes modernos, permite detectar el punto en el que una variable debería dejar de usarse y liberarla de la memoria. La diferencia es que Rust realiza los chequeos necesarios de manera estática, lo que brinda un rendimiento y capacidad de identificar problemas mayor. El encargado de verificar que estas reglas se cumplan, es el *borrow-checker*, que cuando detecta un incumplimiento de las mismas, no permite que el programa compile e informa del problema encontrado.

Las reglas de *ownership* o pertenencia son las siguientes:

- Cada valor en Rust debe tener un dueño.
- Solo puede haber un dueño en cada momento.
- Cuando el dueño del valor sale de su alcance, el valor será eliminado.

Podemos ilustrar estas reglas con programas simples para mejorar su entendimiento.

```
pub fn main() {
    let hola1 = String::from("hello");
    let mundo = "world";
    let hola2 = hola1;

    println!("{},␣{}!", hola1, mundo);
}
```

Este programa parece ser un simple ejemplo de un hola mundo, pero tiene un detalle que lo hace diferente. Una vez dentro de *main*, las primeras dos lineas son declaraciones de las variables hola1 y mundo que contienen Strings con sus equivalentes en texto. En la tercera linea creamos una nueva variable *hola2* a la que le asignamos hola1, para luego en la ultima linea mostrar el contenido de las dos primeras variables. En otros lenguajes, este seria un programa válido, y mostraría por pantalla la "hello, world!". Sin embargo, cuando queremos compilar el código nos da un error.

```
error[E0382]: borrow of moved value: ‘hola1‘
 --> hello_world.rs:6:25
  |
2 |     let hola1 = String::from("hello");
  |         ----- move occurs because ‘hola1‘ has type ‘String‘, which does not implement
    the ‘Copy‘ trait
3 |     let mundo = "world";
4 |     let hola2 = hola1;
  |                 ----- value moved here
5 |
6 |     println!("{},␣{}!", hola1, mundo);
  |                          ^^^^^ value borrowed here after move
```

El compilador es bastante informativo, y nos brinda mucha información sobre el problema. La equivocación esta en que no aplicamos bien las reglas de *ownership*. Cuando declaramos e inicializamos *hola2*, estamos haciendo **uso** de la variable hola1. Al hacer esto, el dueño del String "hello" pasa de hola1 a *hola2* y por lo tanto cuando queremos volver a hacer uso de hola1 para mostrar por pantalla, como solo puede haber un dueño y ese es *hola2*, se produce un error. Este se podría solucionar de algunas maneras: utilizando *hola2* dentro de la macro println!, o realizando explícitamente una copia profunda del valor de hola1 mediante la función clone() o haciendo uso de un *borrow* o préstamo de la variable.

Cada vez que se hace un **uso** de alguna variable, esta misma sale de alcance y volver a utilizarla generaría un error como el que vimos. Es decir, el alcance de una variable inicia cuando esta es declarada y finaliza cuando es **utilizada** o su *scope* acaba. Esto podemos apreciarlo en el siguiente código.

```
    {                           // var no es valida aqui, no fue declarada todavia
        let mut x = 5;
```

```
        let var = "hello";      // var comienza a ser valido aqui.

        x = x * 2
        // hacer cosas con var
    }                           // el scope termina, y por lo tanto var no es mas valida
```

La mecánica de pasar un valor a una función es muy similar a cuando le asignamos un valor a una variable. Al pasar utilizar una variable como argumento de una función va a mover (y transferir el ownership) o copiar el valor automáticamente (por ejemplo con tipos básicos como integer, bool o char), de igual manera que en una asignación. En el ejemplo a continuación, proveniente del libro de Rust (Klabnik and Nichols, 2019), podemos ver como las variables entran y salen del scope al interactuar con funciones con las anotaciones.

```
  fn main() {
    let s1 = dar_ownership();           // dar_ownership mueve su valor de retorno
                                        // dentro de s1

    let s2 = String::from("hello");     // s2 entra en el scope

    let s3 = toma_y_da_ownership(s2);   // s2 es movida a
                                        // toma_y_da_ownership, la cual tambien
                                        // mueve su valor de retorno a s3
  } // Aqui, s3 sale de su scope y es dropeada(eliminada de memoria). Como s2 fue movida
    // no pasa nada. termina el alcance de s1 y es borrada de la memoria.

fn dar_ownership() -> String {              // dar_ownership va a mover su valor de
                                            // retorno dentro de la variable
                                            // llame a esta funcion.

    let some_string = String::from("yours"); // some_string comienza su alcance

    some_string                             // some_string is devuelta y
                                            // movida a donde la funcion fue llamada
}

// Esta funcion toma un String y devuelve el mismo
fn toma_y_da_ownership(a_string: String) -> String { // a_string comienza su scope

    a_string  // a_string es retornada y se mueve a la funcion que realizo la llamada
}
```

La transferencia de *ownership* sigue el mismo patron siempre. Y si bien esto funciona excelente, tomar y transferir el ownership de una variable cada vez que llamamos a algún método puede resultar tedioso y dificultar el uso de la información obtenida dentro del cuerpo de la función. Es por esto que surge la necesidad del concepto de *borrow* o préstamo de una variable, que mencionamos anteriormente como parte de las soluciones al error del compilador.

Una variable propietaria de un valor puede prestar(*borrow*) el acceso a este dato, sin perder su *ownership*, a otras variables mediante el uso de referencias. Estas se designan utilizando el operador "&". Las referencias tienen un comportamiento muy similar al de los punteros, ya que es una dirección de memoria a la cual podemos acceder para obtener información la cual pertenece a una variable. A diferencia de los punteros convencionales, Rust asegura que las referencias siempre van a apuntar a un valor válido de un tipo especifico durante todo su tiempo de vida. Podemos apreciar el uso de *borrow* en el siguiente ejemplo:

```
fn main() {
  let s1 = String::from("mundo");       // creacion de variable s1
  let mut s2 = String::from("hola,␣");  // creacion de variable mutable s2

  add_s1_into_s2(&s1, &mut s2);         // llamada a la funcion add_s1_into_s2 utilizando
    referencias
```

```
  println!("{}", s2);                    // como usamos referencias, ahora podemos mostrar
    s2 sin problemas
}

fn add_s1_into_s2(arg1: &String, arg2: &mut String) {
 arg2.push_str(arg1);                   // hacemos uso de las referencias para concatenar
}
```

El alcance de los argumentos de la funcion es el mismo, pero los valores señalados por las referencias no se eliminan cuando dejan de usarse o salen del scope de la función porque se no tiene propiedad. Cuando las funciones tienen referencias como parámetros en lugar de los valores reales, no necesitaremos devolver los valores para devolver la propiedad, porque esta nunca se tuvo.

Las variables en rust por defecto son inmutables, es decir, que no se pueden realizar modificaciones a los valores o datos de estas una vez declaradas. Lo mismo sucede con las referencias, al momento de declararlas por defecto son solamente de lectura. Esta es una de las tantas decisiones del diseño de Rust que permiten mejorar la seguridad y aumentar la facilidad de escritura y comprobación de código concurrente.

Sin embargo, uno puede declarar explícitamente que una variable o referencia sea mutable, es decir, que permita la escritura de diferentes valores (todos de un mismo tipo) en diferentes puntos del programa. Para esto, se usa la palabra reservada **mut**. En los ejemplos anteriores podemos ver el uso de variables inmutables y mutables en conjunto.

En el ejemplo anterior también podemos apreciar también el uso de la mutabilidad e inmutabilidad de las variables y referencias.

Al igual que el *ownership*, los prestamos y referencias también tienen reglas que debemos seguir. Estas son:

- En cualquier momento, podes tener unicamente solo una referencia mutable o cualquier numero de referencias inmutables
- Las referencias siempre deben ser válidas

De la misma manera, el *borrow-checker* se encarga de comprobar que estas se cumplan a la hora de compilar el programa; y en caso de que falle alguno de los chequeos interrumpir evitar la generación del programa y mostrar el error encontrado. Con estas reglas se pueden evitar muchos problemas, especialmente las condiciones de carrera ya que estas pueden encontrarse cuando dos o mas punteros acceden a una misma dirección de memoria simultáneamente, en donde al menos uno esta escribiendo, y las operaciones no están sincronizadas. Podes tener muchas referencias inmutables, pero solo un puntero mutable, por lo que estas condiciones se detectan y solucionan en tiempo de compilación. Sin embargo, no todo es positivo, ya que si bien hace que la seguridad general de los programas sea muy alta, aumenta mucho la complejidad de implementar ciertos programas. Esto como dijimos en la introducción, abre camino al surgimiento de Unsafe Rust.

1.2.2. Bases de Unsafe Rust

El sistema de ownership de Rust es muy estricto en como las variables pueden acceder a la memoria, esto como vimos da muchas garantías de seguridad. Sin embargo, implementar estructuras de datos que requieran aliasing como listas doblemente encadenadas o grafos resulta demasiado complejo. Mediante el uso de la palabra reservada “unsafe” se permite delimitar un bloque, una funcion o una trait y deshabilitar algunos chequeos del *borrow-checker*. Las únicas cosas diferentes que se pueden hacer en Unsafe Rust, definidas en el Rustonomicon por Beingessner and Klabnik (2016), son:

- Dereferenciar punteros *raw*.
- Llamar funciones unsafe (incluyendo funciones externas, intrínsecos del compilador y el *raw allocator*).
- Implementar traits unsafe.
- Mutar variables estáticas.
- Acceder a los campos de uniones.

El mal uso de alguna de estas operaciones puede causar comportamiento no definido y por lo tanto, comprometer la seguridad y estabilidad del programa. Los comportamientos indefinidos que especialmente debemos evitar al implementar código utilizando unsafe rust son los siguientes:

- Condiciones de carrera.
- Dereferenciar un puntero nulo o colgado.
- Lectura de memoria no inicializada
- Violación de las reglas de aliasing de apuntadores a través de punteros *raw*.
- Valores inválidos en tipos primitivos o en campos/variables locales.

Un ejemplo donde podemos ver el uso de unsafe, extraído desde la librería estándar de Rust, es el siguiente:

```
/// Dereference the given pointer.
/// `ptr` must be aligned and must not be dangling.
unsafe fn deref_unchecked(ptr: *const i32) -> i32 {
    *ptr
}

pub fn main() {
  let a = 3;
  let b = &a as *const _;
  // SAFETY: `a` has not been dropped and references are always aligned,
  // so `b` is a valid address.
  unsafe { assert_eq!(*b, deref_unchecked(b)); };
}
```

El código intenta comparar el valor apuntado por la referencia *raw* "b", que apunta a la variable "a", dos veces obteniéndola directamente y mediante el uso de una función. Declarar un puntero plano o raw es considerado seguro por Rust, sin embargo, no es el mismo caso para acceder a su valor. Por eso la funcion *deref_unchecked* que devuelve el valor apuntado por una referencia plana debe ser marcada como **unsafe**. De la misma manera, al derefenciar "b" y hacer uso de una funcion unsafe el bloque de *main* debe ser delimitado por la palabra reservada **unsafe**.

La comunidad de Rust para mantener los estándares de seguridad, se puso de acuerdo y estableció una guía de practicas que todos los programadores deberían seguir para reducir los riesgos que surge de reducir las limitaciones del borrow-checker. Estas practicas establecen, según el estudio realizado por Astrauskas et al. (2020), tres principios que son:

1. Código unsafe en rust debe ser utilizado de manera **reducida**, para beneficiarse de las garantías inherentes provenientes del lenguaje en la mayor medida posible.
2. Bloques de código unsafe deben ser **directos** y **auto-contenidos** para minimizar la cantidad de código que los desarrolladores tienen que responder por (por ejemplo mediante revisiones manuales)
3. Código unsafe debe ser bien **encapsulado** a través de abstracciones seguras, por ejemplo, proveyendo librerías que no expongan el uso de unsafe Rust a sus clientes.

Además, el mismo paper establece que los mayores uso de unsafe son para la implementación y uso de estructuras de datos complejas, enfatizar contratos e invariantes, y el uso de funciones foráneas. Si bien la necesidad de crear código unsafe se ve reducida debido a la accesibilidad de librerías publicas que brindan acceso a tipos y operaciones que ya solucionan muchos de los problemas, sigue existiendo una gran cantidad de programadores que utilizan código inseguro. En el año 2020, aproximadamente un 24 % de las 34.000 bibliotecas de Rust publicadas en `crates.io` contenían bloques de código no seguro. Es decir, casi 1 de cada 4 proyectos hicieron uso de Unsafe Rust de alguna u otra manera.

La utilización de herramientas automatizadas para facilitar la comprobación del código que debería ser realizada de manera manual, y por lo tanto sujeta a fallos, ayudaría a minimizar los errores y el tiempo necesario para mantener las garantías de seguridad que uno espera de un programa Rust. Ese es nuestro objetivo, y la herramienta que desarrollamos apunta a resolver principalmente los problemas que surgen al deshabilitar los chequeos de aliasing realizados por el *borrow-checker*.

1.3. Objetivos

Nosotros con este trabajo planteamos el desarrollo de una nueva herramienta, utilizando el MIR (representacion intermedia del compilador) para extender el *borrow-checker* y mediante análisis estáticos tratar de detectar los

problemas que de no haberse deshabilitado los chequeos por defecto de Rust habrían sido captados. Especialmente nos enfocaremos en la detección de aliasing de punteros. Implementamos Stacked Borrows para poder detectar los lifetimes de las variables y comprobar que las referencias que puedan superponerse se usen de una manera correcta. Además, implementamos un analizador de alias interprocedural points-to Hind and Burke (1999) Shapiro and Horwitz (1997) para complementar el análisis anterior y obtener aun mayor información.

Gracias a esto deberíamos ser capaces de encontrar la mayoría de las situaciones en las que punteros acceden al mismo lugar de la memoria, e informarle al programador para que pueda comprobar si el código programado es correcto e intencional. El objetivo es que sea una utilidad que funcione como una extension del compilador y ayude a la detección de punteros que son alias, para asi evitar posteriores problemas generados por esto.

Otra situación que tenemos en cuenta es la comprobación de los casts realizados al utilizar punteros planos. Al deshabilitarse las comprobaciones, se podrían realizar transformaciones de tipos que no tengan la misma representación, y esto es peligroso ya que la información almacenada puede quedar inconsistente y generar comportamiento indeseado. Creemos que este problema es posible de abarcar extendiendo el *borrow-checker*.

Para lograr que funcione como una extension del compilador y poder implementar los analizadores que planteamos, crearemos un programa en el lenguaje Rust, utilizando una version nightly (nightly-2022-01-01). Esto ultimo es porque las versiones nightly contienen bibliotecas extras que no se encuentran en la rama estable. Estas nos permiten interceptar el compilador (especialmente nuestro punto de interés es el MIR) y extenderlo. Además brinda herramientas que son totalmente necesarias y útiles para el desarrollo de los análisis y el programa en general.

Durante el segundo capítulo nos centraremos en detallar el sistema de tipos de rust centrándonos en el ámbito unsafe, hablaremos sobre *traits*, *lifetimes*, *raw-pointers* y todas sus particularidades. Tambien discutiremos los trabajos relacionados que se han realizado sobre este tema. Luego, en la tercer sección, detallaremos los conceptos sobre Stacked Borrows (Jung, 2020) y diferentes análisis estáticos extendidos que realizamos. También trataremos como hicimos para extender el *borrow-checker*. El cuarto capítulo nos basaremos en el proceso de diseño e implementación de la herramienta. Mostraremos parte del código, compartiremos las decisiones de porque esta estructurado de esa manera y donde podemos encontrar programadas los distintos apartados que fuimos desarrollando. Para terminar, el quinto y ultimo apartado, se mostrará ejemplos del funcionamiento de la herramienta para sus distintos usos utilizando diferentes programas pequeños de unsafe Rust pequeños como guía, y además observaremos cual podría ser el futuro del proyecto.

Capítulo 2

El sistema de tipos de Rust

Rust es un lenguaje de tipado estático, es decir, realiza la especificación y comprobación de tipos durante el tiempo de compilación y no mientras el programa se esta ejecutando. Además, provee una herramienta muy potente que es la inferencia de tipos. Gracias a esto no es necesario declarar los tipos de las variables de manera explicita en la mayor parte de los casos, sino que Rust infiere que clase de valores mediante un análisis del contexto durante la compilación del programa. Además, esto ayuda a prevenir errores comunes como los errores de tipo y mejora la seguridad y eficiencia del código.

A pesar de ser un lenguaje de tipado estático, provee a través de la palabra reservada **dyn** la posibilidad de realizar el chequeo de tipos en tiempos de ejecución. Si bien al hacer esto perdemos las ventajas de velocidad y seguridad que Rust brinda al realizar los chequeos estáticos, se otorga mayores libertades a la hora de escribir código. Esto puede ser util especialmente en casos donde cierta información, como por ejemplo el tamaño de un valor, es desconocida durante la compilación.

Los tipos primitivos de Rust son los siguientes:

1. Enteros: Los enteros son tipos numéricos que no tienen parte fraccionaria. En Rust, los enteros se pueden dividir en dos categorías: enteros con signo (signed) y enteros sin signo (unsigned). Los enteros con signo son aquellos que pueden ser positivos, negativos o cero, mientras que los enteros sin signo son aquellos que solo pueden ser positivos o cero. Los enteros pueden ser: u8, u16, u32, u64, u128, i8, i16, i32, i64, i128, isize y usize. El carácter i al comienzo representa a los con signo mientras que el carácter u indica los enteros sin signo, ambos seguidos del tamaño de bits que utilizan.
2. Flotantes: Los flotantes son tipos numéricos que tienen una parte fraccionaria. En Rust, los flotantes se dividen en dos categorías: flotantes de precisión simple (f32) y flotantes de doble precisión (f64).
3. Booleanos: Los booleanos son tipos que solo pueden tener dos valores: true o false. El tipo booleano en Rust es bool.
4. Caracteres: Los caracteres son tipos que representan un solo carácter Unicode. El tipo caracter en Rust es char.
5. Cadenas: Las cadenas son tipos que representan una secuencia de caracteres Unicode. El tipo cadena en Rust es str. Además, hay dos tipos de cadenas adicionales: String, que es una cadena que se puede cambiar, y &str, que es una referencia a una cadena.

Los tipos compuestos son aquellos que pueden agrupar varios valores en un solo tipo. Rust tiene dos tipos compuestos primitivos: tuplas y arreglos. Las tuplas agrupan varios valores en una sola variable donde cada valor puede tener un tipo diferente. Los arreglos son tipos que almacenan una cantidad fija de valores del mismo tipo. A diferencia de las tuplas, todos los valores deben tener el mismo tipo.

2.1. Polimorfismo y Traits

El lenguaje Rust realiza una implementación un poco distinta del paradigma de programación orientada a objetos. Rust se distingue de los lenguajes más puros separando la definición de estructuras de datos y sus características, de la implementación de los métodos de esa misma clase o estructura. Mediante la creación de

structs y **enum** se almacenan los datos, y utilizando los bloques de implementación se les provee de métodos y funciones a estos. Los bloques de implementación se generan mediante la palabra reservada **impl**. Esto lo podemos ver apreciado en el siguiente ejemplo:

```
struct Square {
    length: u32,
    height: u32,
}

impl Square {
    pub fn area(&self) -> u32 {
        self.height * self.length
    }
}
```

Creamos el tipo square (cuadrado) que almacena los valores de su altura y su largo. A partir de esto, podríamos crear un objeto de ese tipo brindando la información necesaria, y luego, mediante la función asociada *area* calcular su área utilizando los valores que tenga guardados al momento de la invocación. Esto lo podemos ver reflejado en el código a continuación.

```
pub fn main() {
    let one = Square { length: 2, height: 2 };
    println!("{}", one.area());
}
```

Rust, a diferencia de otros lenguajes orientados a objetos como Java o C++, no implementa el concepto de Herencia. Este es un mecanismo por el cual un objeto puede heredar elementos de otra definición de objetos, para de esta manera, acceder a los datos e implementaciones de un objeto "padre" sin tener que redefinirlos nuevamente. Este mecanismo esta siendo criticado y dejado de utilizar, ya que a menudo se corre el riesgo de compartir más código del necesario entre las distintas subclases. A su vez, limita la flexibilidad en el diseño y puede provocar errores al llamarse métodos que no apliquen a una subclase. El sistema de tipos de Rust en cambio utiliza el concepto de polimorfismo paramétrico, que refiere a la capacidad de una función de recibir por parámetro valores de diferentes tipos. Además de evitar las desventajas del uso de Herencia, esta decisión permite mayor libertad a la hora de hacer uso de covarianza (hacer uso de un tipo más derivado o especifico) y contravarianza (hacer uso de un tipo menos derivado o especifico).

Para implementar el polimorfismo, Rust toma otro enfoque distinto mediante el uso de genericidad y Traits. Klabnik and Nichols (2019) definen una trait o rasgo define la funcionalidad que un tipo particular tiene y puede compartir con otros. Se utilizan los traits para definir comportamientos compartidos de manera abstracta. Podemos usarlos para especificar un tipo genérico, que puede ser aplicado para todo el que cumpla con los rasgos marcados. Tienen una similitud a las interfaces, pero con algunas diferencias. Los Traits de Rust se acercan más a los type-classes de Haskell que a las interfaces de Java.

El comportamiento de un tipo esta marcado por las funciones que proporciona. Diferentes tipos comparten el comportamiento si se puede llamar las mismas funciones en todos esos tipos. Los traits o rasgos son una manera de agrupar estos métodos y definir el conjunto de comportamiento.

En el ejemplo anterior, la clase Square implementa el método area, y si quisiéramos crear una clase Rectangle también deberíamos implementar exactamente la misma función. En el caso de una implementación de Circle o Circulo, la información que almacena el objeto y el cuerpo de la función seria un poco diferente, pero todas estas clases aunque tengan sus particularidades comparten los rasgos de una Figure o figura geométrica.

Cambiando un poco el código, podemos obtener una implementación que permita hacer uso del Rasgo y de esta manera lograr la que funciones genéricas hagan uso del mismo en vez de clases particulares y obtener los beneficios del polimorfismo.

```
pub trait Figure {
    pub fn area(&self) -> f32;
}
```

```
impl Square for Figure {
    pub fn area(&self) -> f32 {
        self.height * self.length
    }
}
```

En este caso definimos un único método dentro del Trait, pero pueden haber múltiples donde cada encabezado debe estar en una linea diferente y terminar con un punto y coma.

2.2. Lifetimes

Según Klabnik and Nichols un *lifetime* o tiempo de vida es una construcción del compilador (mas específicamente del *borrow checker*) usada para asegurar que todos los prestamos sean validos. El tiempo de vida de una variable comienza cuando es creada y termina cuando es destruida. Mientras que lifetimes y scopes son usualmente referidos juntos, no son lo mismo.

Por ejemplo, en el caso de que tomemos prestada una variable a traves de una referencia. El borrow tiene un lifetime que esta determinado por donde es declarado. Y como resultado, este es válido mientras que termine antes de que el owner sea destruido. En cambio, el scope del préstamo esta determinado por el bloque en el que la referencia es usada.

```
fn main() {
    let i = 3; // Lifetime for `i` starts. ----------------+
    {                                                      //    |
        let borrow1 = &i; `borrow1` lifetime starts. //---+|
        println!("borrow1: {}", borrow1);             //   ||
    } //``borrow1'' ends. ------------------------------+|
    {                                                      //    |
        let borrow2 = &i; `borrow2` lifetime starts. //---+|
        println!("borrow2: {}", borrow2);             //   ||
    } //``borrow2'' ends. ------------------------------+|
}   // Lifetime ends. -------------------------------------+
```

En el ejemplo anterior podemos ver como los scopes y lifetimes están relacionados y se diferencian. Hay que notar también que en el código no se utilizaron nombres o tipos asignados para las etiquetas de los tiempos de vida. Esto es gracias al mecanismo de "**lifetime elision**" que implementa el compilador de Rust, que al igual que con los tipos, es capaz de identificar en gran medida los lifetimes y colocarlos automáticamente sin necesidad de establecer las etiquetas de manera explicita.

Sin embargo, cuando dos o más lifetimes están involucrados es necesario establecer explícitamente que las variables tienen diferentes lifetimes. Para esto se utiliza el operador ´ seguido de una letra. Un ejemplo de esto podría ser el encabezado de la función longest, que compararía dos variables que contengan cadenas, y una de estas podría tener un tiempo de vida mayor que la otra. Esto se representaría de la siguiente manera.

```
    fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &'a &str;
```

Existe un lifetime especial **'static** (estática) que denota que una referencia puede vivir por la duración entera de la ejecución del programa. Es necesario recalcar que su uso debe ser pensado con cuidado, porque podrían generarse referencias colgadas o problemas de compatibilidad entre lifetimes. Generalmente suelen ser utilizadas unicamente para crear constantes globales.

Sin tener en cuenta el mecanismo de **lifetime elision**, las declaraciones de funciones deben cumplir con las siguientes restricciones:

- Toda referencia debe tener un tiempo de vida etiquetado.
- Toda referencia retornada debe tener el mismo tiempo de vida que alguno de sus inputs o ser *static*

Estas reglas también se extienden para las declaraciones de métodos, structs y traits.

2.3. Raw Pointers y Unsafe Rust

Una de las mayores atracciones de Rust son las garantías de seguridad estáticas sobre el manejo de memoria. Sin embargo, los chequeos estáticos son conservadores por naturaleza. Existen programas que son seguros, pero el compilador no puede verificar que lo sean y los rechaza. Además, realizar programaciones de bajo nivel como interacciones directas con partes del sistema operativo o inclusive implementar un sistema operativo son tareas inherentemente riesgosas. Por estos motivos, existe Unsafe Rust. Al utilizar la palabra reservada **unsafe** creamos un nuevo bloque para que contenga las operaciones inseguras, y le comunicamos al compilador que entendemos los riesgos y relaje sus restricciones para el mismo.

Klabnik and Nichols (2019) (capitulo 19) establece que las acciones o *superpoderes* que se pueden hacer en unsafe Rust y no en safe Rust son:

- Acceder o actualizar una variable mutable estática.
- Dereferenciar un apuntador plano (raw pointer).
- Llamar e implementar funciones unsafe.

Y es importante recalcar que utilizar unsafe no desactiva el *borrow-checker* ni ninguno de los otros chequeos de seguridad de Rust: si utilizas una referencia en unsafe rust, se le realizaran los chequeos de igual manera.

No todo lo que este dentro de un bloque unsafe es necesariamente riesgoso o va a acceder de una manera invalida a la memoria. Toda persona esta sujeta a fallos y los errores pueden cometerse, por eso al realizar operaciones inseguras en bloques anotados y delimitados por **unsafe** ayuda al aislamiento y facilidad de encontrar los errores relacionados con los problemas de seguridad de la memoria. Al encapsular código unsafe en bloques pequeños también se evita propagar los fallos a otros bloques o librerías que no estén relacionadas o que se asuman seguras.

Los punteros planos son tipo en Unsafe Rust que son muy similares a las referencias, pero también se diferencian de estas. Los apuntadores planos te permiten llevar a cabo aritmética de punteros arbitraria, y pueden causar un numero de problemas de seguridad. En algunos sentidos, la habilidad de dereferenciar un apuntador arbitrario es una de las cosas mas peligrosas que puedes hacer. A diferencia de las referencias, los punteros planos:

- Permiten ignorar las reglas de borrow (préstamo) al tener punteros mutables e inmutables o multiples punteros mutables en un mismo bloque.
- No garantizan apuntar a un sector valido de la memoria
- Pueden ser null (vacíos o no inicializados)

Al igual que las referencias, los raw pointers pueden ser mutables o inmutables y se escriben ***mut T** y ***const T** respectivamente.

```
let mut value = 1;

let immutable = &value as *const i32;
let mutable = &mut value as *mut i32;

unsafe {
    println!("Inmutable␣es:␣{}␣y␣Mutable␣es:␣{}", *immutable, *mutable);
}
```

En este simple ejemplo podemos ver como se crean punteros planos usando la palabra reservada **as** para hacer un casteo de una referencia mutable o inmutable a su raw pointer correspondiente. También podemos observar que no es necesario crear punteros planos dentro de un bloque unsafe ya que no genera ningún peligro; sin embargo, cuando tratamos de acceder a los valores que apuntan pueden generarse comportamientos inadecuados en el programa y por lo tanto deben enmarcarse en unsafe. En el ejemplo, usamos el operador * para acceder a la información apuntada por una referencia o puntero plano, y asi mostrarla por pantalla usando la macro *println!*.

Podemos notar que ambos punteros mutable e immutable apuntan a la misma dirección de memoria, donde value esta almacenada. Si hubiésemos utilizado referencias, el programa no habría compilado ya que se rompen las reglas del *borrow-checker*. En el ejemplo anterior estamos accediendo a una variable de manera inmutable

y luego podemos potencialmente modificarla mediante el puntero mutable, lo que produciría una condición de carrera.

El siguiente fragmento de código seria rechazado por el compilador si no se hiciera uso del unsafe:

```
let mut num = 3;

let x = &mut num as *mut i32;
let y = &mut num as *mut i32;

unsafe {
    *x = 1;
    *y = 2;
}
```

En este programa, podemos ver que existen dos punteros planos mutables que comparten la misma dirección de memoria, en la que se encuentra el valor de num. Si se hicieran uso de referencias en vez de raw-pointers, se romperían las reglas del *borrow-checker* y por lo tanto el código no compilaría. Vemos que las variables **x** e **y** están ambas modificando el mismo valor. Si no se tiene cuidado, esto puede generar problemas como inconsistencia en la memoria, condiciones de carrera, comportamientos no deseados en el programa, fugas en la memoria, etc.

Aun asi, muchos pueden optar por hacer uso de esta herramienta para implementar programas de una manera mas eficiente o simple. Un ejemplo de esto, mencionado también por el libro de referencia de Rust, son las listas doblemente encadenadas, la cual no tiene una estructura de árbol y es complicado crearlas solo en safe rust. Extendiendo una lista simple utilizando raw-pointers y bloques unsafe para apuntar a nodos anteriores da una solución simple a este problema.

Beingessner explora de manera didáctica la creación de estas listas encadenadas, y los contratiempos que uno se encuentra con los diferentes métodos de implementar estas estructuras. Ella establece que el problema más importante es el aliasing de punteros. Dos punteros son alias cuando apuntan a una misma dirección de memoria. El compilador usa la información acerca de donde referencian los punteros para optimizar los accesos a la memoria, por eso si la información que tiene es errónea entonces el programa va a ser mal compilado y generar resultados aleatorios que no sirven. El compilador debe saber cuando es seguro asumir que valor puede ser recordado"(cache) en vez de ser leido reiteradamente.

2.4. Trabajos relacionados

MIRI Olson (2016) es un interprete experimental de la representación intermedia del nivel medio (MIR) de Rust. Esta desarrollada al mismo tiempo que el compilador, y tiene como objetivo ayudar a encontrar ciertos casos de comportamiento no deseado en los programas de Rust. Entre algunos que podemos mencionar son:

- Acceso de memoria out-of-bounds y use-after-free
- Uso de valores sin inicializar
- Violaciones de los invariantes de tipos básicos
- Fugas de memoria
- etc.

Con esta herramienta no es posible detectar todos los posibles comportamientos no deseados, pero podría considerarse como la herramienta estándar de los programadores Rust para encontrar errores en el código debido a la diversidad de errores importantes que ataca y a su alto nivel de mantenimiento. Ellos creen que el aliasing también es un problema muy importante, y por eso implementan Stacked Borrows Jung (2020) para afrontarlo. Stacked borrows es una semántica operacional para Rust, que define condiciones bien marcadas en las cuales Rust exhibe comportamiento indeseado debido a errores de aliasing.

La idea principal de su implementación es definir una version dinámica del análisis estático (o *borrow-checker*), que Rust ya utiliza para comprobar que las referencias sean accedidas de manera acorde con las reglas que venimos mencionando. A pesar de que esta idea es muy buena, realizar los chequeos de manera dinámica reduce

tiene un efecto en el rendimiento y además la herramienta MIRI requiere de información extra como la existencia de tests o definiciones de pre y post condiciones para poder hacer ciertos tests.

Rudra Yechan and Youngsuk (2021) es un analizador estático el cuál su objetivo principal es la detección de los comportamientos indefinidos comunes en Rust. Es capaz de analizar tanto programas simples como paquetes de cargo. Los bugs detectados por esta herramienta son problemas de Panic Safety, Higher Order Invariant y Send Sync Variance, los cuales se producen cuando se mal utiliza el unsafe en situaciones particulares, como al generarse un panic o realizarse un mal uso de código asíncrono. Si bien su enfoque es distinto, al utilizar análisis estático es muy eficiente y esto se ve evidenciado en que fue capaz de analizar la totalidad de los paquetes publicados en crates.io en menos de 7 horas.

Otro proyecto a tener en cuanta es MirChecker Li et al. (2021), que al igual que Rudra y este trabajo, utiliza un análisis estático del MIR para crear un framework automatizado para la detección de bugs. Realizan una cantidad muy grande de tests en donde comprueban errores matemáticos, lógicos, loops infinitos, accesos fuera de rangos, usos después de liberación de memoria, etc. Esta herramienta no es tan utilizada como MIRI o Rudra, pero al ser muy diversa es útil para detectar muchos errores.

Todos estos proyectos tratan de enfocar los conflictos que pueden encontrarse al hacer uso de Unsafe Rust, e inclusive utilizan métodos y enfoques similares, pero cada uno apunta a resolver obstáculos diferentes. Este proyecto toma en cuenta las bases sentadas por estas herramientas, y nos enfocaremos a atacar problemas especificos diferentes que creemos que son igual de importantes, y esperamos que el uso conjunto con alguna de las anteriores permita abarcar y encontrar una mayor cantidad de errores de código.

Capítulo 3

Extendiendo el borrow checker

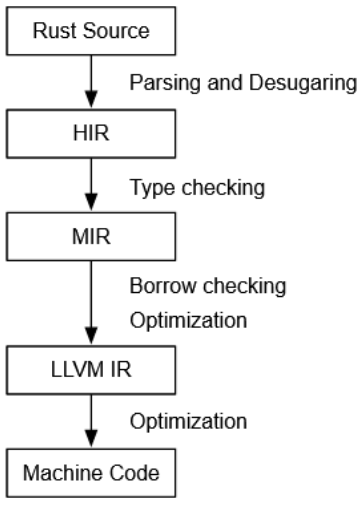
3.1. Etapas del proceso de compilación

El compilador de Rust según M (2022) se diferencia del resto, ya que hace procesos que otros no realizan (borrow-checking) y también toma algunas decisiones no convencionales. El proceso de compilación de un programa comienza al invocar el comando **rustc** junto al código fuente de un programa y los parámetros. El modulo *rustc_driver* se encarga de captar los argumentos y definir la configuración que se utilizará en el resto del procedimiento.

Una vez que el compilador captó el código, los módulos *rustc_lexer* y *rustc_parse* se encargan de analizarlo y generar el AST (árbol de sintaxis abstracta). En esta primera etapa también se expanden las macros, se resuelven los nombres, se chequea la sintaxis del programa y se valida que el AST sea correcto.

Teniendo este árbol, se proceden a generar las representaciones intermedias. La primera en construirse es el HIR (High-Level Intermediate Representation) que es una version más amigable para el compilador del AST. Durante este proceso se realiza la inferencia de tipos, la resolución de las traits y principalmente el chequeo de tipos, donde se convierten los tipos encontrados en el HIR por representaciones internas usadas por el compilador. Estas son usadas para aumentar la seguridad, correctitud y coherencia de los tipos utilizados en el programa.

Figura 3.1: Compiler pipeline



La siguiente parte es la construcción del MIR (Mid-Level Intermediate Representation) a partir del HIR. Aquí es donde se hacen gran parte de las optimizaciones del código, junto al pattern-matching y chequeos de exhaustividad. Además, el MIR es donde se realiza el borrow-checking y otros chequeos importantes basados en el flujo de datos. Debido a que es una representación de alto nivel y genérica es aquí donde se realizan la mayoría de análisis del compilador, y nosotros tomaremos en cuenta esto para añadir nuestros propios algoritmos y análisis.

A continuación se realiza lo que es conocido por la generación de código o *codegen*, que consiste en transformar las representaciones de alto nivel a un binario executable. **rustc** utiliza **LLVM** para esto. Se comienza convirtiendo el MIR en LLVM IR (LLVM Intermediate Representation) para luego pasárselo a LLVM, el cual realiza más optimizaciones, y emitir el código de maquina. Este es básicamente código assembler con algunos tipos y anotaciones añadidos. Los diferentes binarios/bibliotecas luego son unidos (linked) para crear el binario final.

3.2. Análisis estáticos

3.2.1. MIR

M (2022) dice el MIR que es una forma simplificada de Rust el compilador utiliza principalmente para comprobaciones de seguridad flow-sensitive, como por ejemplo el borrow checker. Esta definido en el modulo

rust_middle::mir, se basa en un CFG (Control-Flow Graph o Grafo de Control de Flujo) y todos los tipos son explícitos.

El CFG es un término común en el ámbito de los compiladores, ya que permite representar un programa exponiendo de manera clara el flujo del mismo. El CFG del MIR esta estructurado como un conjunto de bloques básicos conectados por aristas. Estos bloques básicos están conformados por un conjunto de *statements* que se ejecutarían juntos y de manera secuencial y completa. Al final se encuentra un *terminator* cuya función es referenciar y conectar bloques básicos. La construcción de un CFG es generalmente el primer paso de la mayoría de los algoritmos de análisis flow-sensitive, y el MIR esta estructurado de esta manera para facilitar estos análisis.

Otros conceptos claves del MIR son:

- **Locals**: Lugar de la memoria alojado (conceptualmente) en el stack. Se representan con un guion bajo seguido de un numero, por ejemplo _1. El lugar _0 esta reservado para la dirección de retorno de la función.
- **Places**: Expresiones que identifican un lugar en la memoria.
- **Rvalues**: Expresiones que producen un valor. Su nombre proviene del hecho de que están del lado derecho de una asignación.
    - **Operands**: Son los argumentos de un rvalue, que pueden ser constantes o Places

Mediante el uso de las versiones nightly del compilador, que añaden la posibilidad de importar módulos críticos e imperativos para los análisis estáticos que queremos realizar, es posible acceder y utilizar las distintas representaciones intermedias de un código fuente.

Podemos apreciar la transformación a MIR mediante un ejemplo. El siguiente programa básico

```
fn main() {
    let mut x = 5;
    x = x + 10;
}
```

tiene una representación en MIR de la siguiente forma:

```
// WARNING: This output format is intended for human consumers only
// and is subject to change without notice. Knock yourself out.
fn main() -> () {
    let mut _0: ();              // return place in scope 0 at ./examples/simple_sum.rs
:1:11: 1:11
    let mut _1: i32;             // in scope 0 at ./examples/simple_sum.rs:2:9: 2:14
    scope 1 {
        debug x => _1;                   // in scope 1 at ./examples/simple_sum.rs
:2:9: 2:14
    }

    bb0: {
        _1 = const 5_i32;        // scope 0 at ./examples/simple_sum.rs:2:17: 2:18
        _1 = const 15_i32;       // scope 1 at ./examples/simple_sum.rs:3:5: 3:15
        return;                  // scope 0 at ./examples/simple_sum.rs:4:2: 4:2
    }
}
```

Este ejemplo, como indica el comentario arriba, muestra el MIR para que sea más sencillo de leer por las personas. Podemos apreciar como se crea la función main, se realizan las declaraciones de las variables en la parte superior y luego procede a definirse los bloques básicos. En este caso, el bloque bb0 es el único que se construyo y contiene dos statements y un terminator. Debido a la optimizaciones del compilador (propagación de constantes), los statements son solo asignaciones directas sin la necesidad de hacer una suma como en el programa fuente. El terminator en este caso es la llamada a return.

Para extender el borrow checker, se realiza una interpretación abstracta del MIR aplicando los análisis estáticos necesarios y recopilando la información.

3.2.2. Stacked borrows

Stacked Borrows Jung (2020) propone una semántica operacional para los accesos de memoria en Rust que refuerza. Stacked borrows define una disciplina de aliasing y declara que cualquier programa que la viole contendrá comportamiento no definido, lo que significa que el compilador puede descartarlos a la hora de realizar optimizaciones. Esto lo realiza introduciendo condiciones claramente definidas en las cuales un programa Rust no se comporta debidamente debido a un error de aliasing. La idea es definir una version dinámica del *borrow-checker* que Rust ya utiliza para comprobar que los accesos se hagan de acuerdo a la política de aliasing.

El borrow checker en particular comprueba que las referencias que puedan superponerse, se utilicen de una manera bien anidada. Stacked Borrows modela esta disciplina mediante un análisis haciendo uso de una pila por locación: se detecta cuando las referencias no son utilizadas que sigan el método de pila, y se marca a esos programas como erroneos. Basados en eso, se extiende Stacked Borrows para las reglas de los punteros planos (que son ignorados por el borrow-checker) con el objetivo de ser lo más liberes posibles sin interferir con las propiedades principales de un "fragmento seguro"del análisis que realizan.

La idea principal detrás de Stacked Borrows es tomar el análisis estático que realiza el borrow checker, el cual utiliza lifetimes, y transformarlo en un análisis dinámico el cual no haga uso de lifetimes. De esta manera, se puede lograr que inclusive aquellos programas que contengan unsafe deban satisfacer esta comprobación. Los programas Safe deberían trivialmente satisfacer el nuevo chequeo, ya que este es estrictamente mas libre que el anterior.

De manera simplificada, la version dinámica del borrow checker debería asegurar que:

1. Una referencia y todas las referencias derivadas de esta puedan ser usadas solamente durante su lifetime, y
2. El referente no es utilizado hasta que el lifetime del préstamo haya expirado.

Consideremos el ejemplo brindado por Jung (2020):

```
1 let mut local = 0;
2 let x = & mut local ;
3 let y = & mut *x;   // Reborrow x to y.
4 *x = 1;             // Use x again.
5 *y = 2;             // Error ! y used after x got used
```

Este programa viola el principio de la debido a un uso de la referencia y que ocurre en la linea 5 despues del siguiente uso del referente x en la linea 4, y por lo tanto este programa es rechazado por el borrow-checker. Si observamos el patron de uso, podemos observar que "XYXY" es una violación a la idea de anidación. Para que el programa sea conforme con la política de la pila, debería tener los accesos anidados de la manera "XYYX".

Para llevar esta idea a cabo, Stacked Borrows define un modelo operacional. Primero se debe ser capaz de distinguir las referencias que apuntan a un mismo lugar de la memoria; por eso se asume que todas las referencias son marcadas con un ID único cuando son creadas, y este se preserva a medida que la referencia es copiada. Luego, en memoria se debe almacenar una pila que almacenará los IDs de las referencias junto a información para distinguir el tipo de item (Unique, SharedReadOnly o SharedReadWrite) que es.

Las principales reglas del modelo son las siguientes:

- (new-mutable-ref) Cada vez que una nueva referencia mutable es creada (&mut expr) de algún valor referente existente, primero que nada se considera *uso* de ese referente. Luego se elige un ID nuevo para la referencia, y se la coloca junto a su tipo (Unique) en el tope de la pila.
- (new-mutable-raw) Cada vez que una variable mutable plana (raw pointer) es creada mediante un casteo (expr as mut T) a partir de una referencia mutable (&mut T) con el valor de algun referente, primero es considerada un uso de esa referencia mutable. Luego se añade la referencia del tipo SharedReadWrite al tope de la pila.
- (new-shared-ref) Cada vez que una nueva referencia compartida es creada (&expr) a partir de un valor existente de un referente, primero es considerado un *acceso de lectura* del valor. Luego se elige un ID nuevo para la referencia, y se la coloca junto a su tipo SharedReadOnly en el tope de la pila.
- (uso) Cada vez que un referente X es utilizado, un item con el ID X debe estar en la pila. Si existen otros items por encima de este, hay que removerlos, de tal manera que X quede en el tope de la pila. En

el caso de que la pila no contenga ningún item Unique con el mismo ID del referente usado, o un valor SharedReadOnly o SharedReadWrite, entonces el programa tiene comportamiento no deseado.

- (lectura) Cada vez que una referencia X es leida, un item con el ID X debe encontrarse en la pila. Se remueven los elementos del tope de la pila hasta que todos los items por encima de X son SharedReadOnly. Si no existe ningún item con ID X en la pila, el programa viola el modelo planteado.

Existen ciertas extensiones y optimizaciones que pueden realizarse a este modelo, sin embargo estas son las reglas que consideramos mínimas para detectar errores de aliasing en la mayor parte de programas y son las que implementaremos en la herramienta.

Stacked borrows fue definido formalmente y esta verificado utilizando Coq. Además, como hemos mencionado el interprete MIRI Olson (2016) dentro de las variedades de chequeos que realiza, posee una implementación bastante robusta del modelo de Stacked Borrows donde se pueden realizar tests para comprobar su utilidad.

3.2.3. Análisis points-to

Existen un gran número de estudios y algoritmos realizados para el análisis del flujo de datos en los programas, y en especial para computar la información points-to. Shapiro and Horwitz (1997) establece que para realizar análisis en un programa que involucra punteros, es necesario tener información (segura) acerca de a donde apunta cada puntero. En general, mientras mas precisa se tenga la información de a donde apunta (points-to), más preciso van a ser los resultados obtenidos. Los análisis de flujo de datos como propagación de constantes, variables vivas, comprobaciones de alcance, etc. necesitan saber a donde una variable podría estar apuntando en cada declaración y operación.

Hind and Burke (1999) define que un Alias se produce cuando existe mas de un camino de acceso a un lugar en la memoria. Un camino de acceso es un identificador que apunta a un lugar especifico en la memoria, estos pueden ser punteros, expresiones conformadas por variables, etc. Dos caminos son must-alias si en una declaración S, si refieren a la misma locación en memoria en todas las instancias de ejecución de S. Dos caminos son may-alias en S si refieren a la misma locación en memoria en alguna de las ejecuciones de S, es decir, may-alias establece que pueden ser alias pero no lo asegura. La computación de may-aliases incluye los must-aliases como subconjuto. Los algoritmos a utilizar realizaran el calculo de los may-aliases y por el motivo anterior solamente se los mencionaran como alias.

Aldrich (2017) define dos tipos comunes de análisis de punteros, que son análisis de alias y de points-to. El análisis de alias computa un conjunto S que contenga los pares de variables (p,q) donde p y q pueden (o deben) apuntar a un mismo lugar en memoria. En cambio, el análisis points-to computa las relaciones apunta-a(p,x) donde p puede (o debe) apuntar a la locación de memoria de la variable x. Si bien ambos tienen el mismo objetivo, los métodos y algoritmos difieren. Nosotros utilizaremos un análisis points-to basado en la idea originalmente propuesta por Andersen, que es una de las más reconocidas y utilizadas para este propósito.

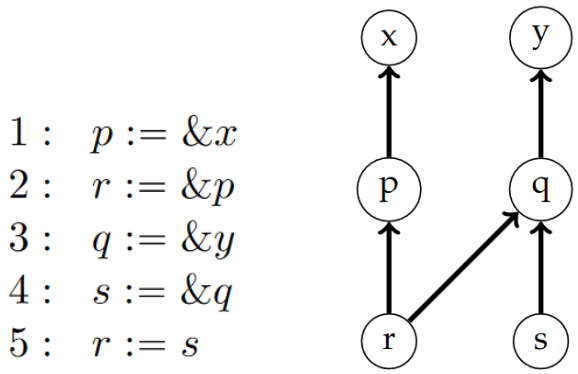
El análisis points-to de Andersen es interprocedural, y no tiene en cuenta el contexto o flujo. Esto quiere decir que no tiene en cuenta el orden de las declaraciones del programa, y esto se realiza para mejorar el rendimiento ya que estos tipos de estudios pueden ser muy costosos en practica. Sin embargo, para mejorar la precisión y debido a que el MIR es generado de manera secuencial no haremos uso de esta característica.

Para realizar este análisis, creamos un grafo dirigido para la representación de los aliases. Creamos un nodo para cada variable del programa, y haciendo una lectura secuencial de las declaraciones unimos utilizando aristas dirigidas cada vez que una variable X referencia a un valor apuntado por otra variable Y. De esta manera obtenemos todos los pares points-to(X,Y) donde X apunta a Y, y se ve reflejado en el grafo.

Este comportamiento lo podemos apreciar en el ejemplo brindado por Aldrich (2017), observando el pseudo-programa y el grafo de Andersen construido.

Una vez obtenido el grafo de points-to es posible saber cuales variables son alias. Aquellos que nodos que tengan más de una arista entrante indican que existe más de un camino para acceder al valor ubicado en ese lugar de la memoria, y por lo tanto, se produce alias. Además, si utilizamos estos datos en conjunto la información obtenida de Stacked Borrows simulando los lifetimes de las variables, podemos añadir precisión al análisis. Esto se debe a que al realizar points-to sabemos que puede ser alias, pero no es correspondiente asegurarlo. Sin embargo, gracias a Stacked Borrows podemos saber si algunas de las referencias involucradas en el alias dejaron de utilizarse y por lo tanto, indicar una mayor posibilidad de asegurar o descartar el problema de aliasing.

Si bien este método es considerado interprocedural, es necesario tener en cuenta los aspectos mencionados por



(a) Pseudocodigo (b) Grafo

Figura 3.2: Creación de grafo points-to

Hind and Burke (1999). Para que un análisis de flujo de datos sea interprocedural, se debe hacer uso del PCG (Program Call Graph), que es un multi-grafo en el cual cada procedimiento o función se representa como un único nodo y una arista (f,g) representa una potencial llamada a la función g desde el procedimiento f. Si bien en el HIR es más simple obtener y navegar el PCG que se describe, se añadiría una complejidad extra al manejar ambas representaciones intermedias simultáneamente. Por eso, para reemplazar el uso del PCG se utilizan los *terminator* del MIR. Estos nos brindan toda la información necesaria de las diferentes llamadas a procedimientos externos, y nos permiten mediante el estudio de los argumentos actuales y formales asociar las variables y sus lugares compartidos en la memoria entre funciones. De esta manera además ganamos eficiencia y precision, ya que solamente tenemos en cuenta las llamadas a funciones que unicamente están relacionadas con el procedimiento que queremos analizar y que realmente sean utilizadas.

3.3. Otros análisis

Además de los análisis anteriormente mencionados, es posible realizar muchos mas gracias a la gran cantidad de información presente en el MIR sin realizar grandes cambios a la estructura del proyecto. Nosotros consideramos agregar un estudio de los casts realizados de manera unsafe, en especial nos centramos en encontrar los errores que puedan surgir de hacer transformaciones de un tipo que contenga una representación de X cantidad de bits, a una que posea una cantidad diferente.

Debido a que el borrow checker esta desactivado durante la creación de raw pointers, es posible hacer casts de tipos las restricciones convencionales. Por ejemplo, se podría intentar trasformar un flotante f32 que necesita mínimo de 32 bits, utilizando un puntero plano, a un entero u16 que necesita mínimo de 16 bits. Esto claramente es un problema, ya que si intentamos utilizar este entero es muy probable que se produzca comportamiento no deseado, ya que estamos quitándole arbitrariamente la mitad de la información del tipo original sin hacer ningún chequeo de si el resultado sea correcto.

Para realizar esta comprobación simplemente recorremos el normalmente MIR hasta encontrar una sentencia de Cast. Una vez que se encuentre, observamos las variables involucradas y obtenemos la información de sus respectivas representaciones. Con estos datos, procedemos a comparar estas representaciones en base a la cantidad de bytes que posean, y si encontramos que no poseen la misma cantidad y que se esta intentando transformar una representación de mayor cantidad de datos a de menor cantidad, se procede a informar de la situación. De esta manera, el programador podrá ver el aviso y tomar los recaudos necesarios para asegurarse de que este problema no rompa la política de seguridad de Rust.

Capítulo 4

La herramienta: diseño e implementación

4.1. Diseño

La arquitectura de nuestro programa esta fuertemente inspirada en el diseño de las herramientas de Rudra Yechan and Youngsuk (2021) y MirChecker Li et al. (2021). Podemos dividir el proceso en tres etapas: Interfaz de usuario, Análisis estático y Comprobaciones finales.

1. Interfaz de Usuario: En esta primera etapa se realiza la captura del código fuente del programa Rust que se desea analizar junto con las opciones a tener en cuenta, como por ejemplo, la función de entrada en caso de que no sea la principal. Para esto, se pueden utilizar dos formas. Una de ellas es ejecutar es compilar el proyecto y utilizarlo desde la llamada "cargo run" seguido del nombre del archivo .rs y opcionalmente el nombre de la función inicial. Esta es la forma mas sencilla y rápida para ejecutarla en archivos individuales. La otra posibilidad es instalar el proyecto como un subcomando del paquete cargo, y de esta manera llamar a la herramienta dentro de otro paquete. La ventaja de esto es que permitiría hacer análisis entre distintos archivos, y además se facilitaría el uso de otras herramientas del ecosistema Rust. Una vez capturada el código fuente del procedimiento a analizar, se genera el MIR optimizado del mismo junto con sus llamadas a funciones en caso de que existan y se prosigue a la siguiente etapa.

2. Análisis estáticos: Detrás de la interfaz de usuario, se encuentra el analizador estático implementado como una modificación del compilador de Rust con análisis adicionales. La segunda etapa comienza con el estudio del MIR, donde iterativamente se realiza una interpretación abstracta para cada uno de sus bloques básicos y se almacena en memoria todos los datos relacionados con los análisis a realizar. Estos datos pueden ser utilizados inmediatamente durante esta etapa para procesar resultados intermedios, o ser de utilidad en la siguiente etapa para sacar conclusiones finales post-análisis. El estudio del MIR se realiza mediante la implementación del patron de diseño Visitor Presman (1997), que se encuentra definido como una Trait dentro del propio compilador de Rust en el paquete **rustc_middle::mir::visit**. Este patron permite separar los algoritmos de la estructura del objeto en el que opera. Generalmente se utiliza para agregar nuevas operaciones a estructuras existentes sin modificarlas. Nosotros lo utilizaremos para navegar en el MIR, visitando cada uno de los bloques, sentencias y terminadores; y utilizar los datos de estas para realizar nuestros análisis. Dentro de esta etapa se realizara los análisis de casteos, del perfil de funciones, de Stacked Borrows y se comenzará a generar el grafo de Andersen para el análisis points-to de aliasing. Es posible que durante las comprobaciones se descubran errores de Stacked Borrows o Casteos, los cuales serán informados inmediatamente en pantalla para que sea más simple ubicar el error dentro del MIR y por lo tanto, dentro del código base.

3. Comprobaciones finales: Terminada la interpretación del MIR, se procede a realizar estudios con los datos obtenidos durante la etapa anterior para determinar potenciales errores, e informar mediante mensajes generados acordes al problema encontrado en caso de que exista. Brindamos en formato .dot el grafo de Andersen, para que sea posible visualizar el mismo mediante el uso de herramientas como Graphviz. Además, procedemos a realizar el análisis points-to, que junto a los lifetimes obtenidos por Stacked Borrows podemos llegar a la conclusión de si posiblemente existe alias o no en alguna de las variables utilizadas y consecuentemente mostrar el resultado por pantalla.

Este diseño permite lograr una gran escalabilidad, debido a la velocidad que se obtiene del enfoque estático de los análisis y a la modularidad que permitiría acoplar nuevos análisis sin realizar demasiados cambios a la estructura del proyecto. Además, gracias a trabajar con el MIR podemos hacer estudios con tipos genéricos sin conocimiento de las formas concretas de sus parámetros, lo cual no seria posible si se utilizara un análisis de una representación de bajo nivel como por ejemplo LLVM IR.

4.2. Implementación

La herramienta esta implementada totalmente en Rust, utilizando la version *nightly-2022-01-01*. El principal analizador esta implementado como un driver personalizado del compilador de Rust. Trabaja como un compilador de Rust sin modificar al compilar las dependencias e injecta los algoritmos de análisis cuando se compila el paquete a estudiar. También coopera con Cargo, el manager de paquetes oficial de Rust, para brindar una experiencia de uso similar al resto de las herramientas integradas con el mismo.

Dentro de la carpeta base del proyecto (*/src*) podemos encontrar archivos relacionados con los análisis principales a realizar como stacked borrows y points-to, junto a dos carpetas *bin* y *mir_visitor*. Dentro de *bin* podemos encontrar la sección de Interfaz de Usuario, tenemos dos archivos donde cada uno se encarga de tomar el código fuente de un programa Rust y brindarle la información al analizador principal. Cada uno de estos archivos intenta obtener el código fuente de una manera distinta, uno lo hace directamente mientras que el otro utiliza cargo como intermediario para buscar dentro de un paquete.

El analizador principal se encuentra dentro del archivo *analyzer.rs*. Este se encarga de obtener MIR de la función a analizar, y en caso de que exista procede a realizar la interpretación abstracta de este. Este proceso comienza al crear un Visitor partiendo del Body del procedimiento principal y llamar a la función *visit* que se encargará de acceder secuencialmente a cada una de las partes de la representación intermedia. El patrón visitor necesario para recorrer el MIR se encuentra implementado dentro de la carpeta *mir_visitor*. Allí se encuentran 4 archivos, de los cuales 3 están destinados a la interpretación y análisis de distintas partes (Bloques, Cuerpo y Terminadores) del MIR y un cuarto archivo con funciones de ayuda que es utilizada por los anteriores.

La interpretación comienza primero por el cuerpo de la función, donde tenemos ya información del nombre de la misma, y de sus variables. Dentro de *visit_body* estudiamos las declaraciones locales y obtenemos todos los bloques básicos con sus datos. Luego, llamamos a *visit_basic_block_data* para cada uno de los bloques y respetando el orden. Adentro de los bloques, obtenemos la información de cada una de las sentencias y la existencia de algún terminator. Con esto, invocamos secuencialmente *visit_statement* donde visitaremos todas las sentencias de código del bloque. Estas pueden ser de varios tipos, pero el más común e importante son las asignaciones. Mediante *visit_assign* podemos saber si estas asignaciones corresponden a casteos, a referencias, a operaciones binarias o unarias, a creaciones de variables mutables o inmutables, etc. y con estos datos actualizar la información correspondiente a los análisis de **points-to** y **Stacked Borrows**. Durante cada asignación, además de actualizar los datos en memoria procedemos a imprimir por pantalla los datos del MIR para facilitar la visualización y detección de problemas. Finalizada la interpretación de cada una de las sentencias, procedemos a realizar la visita a los terminadores mediante *visit_terminator*. Al igual que las asignaciones, existen varios tipos de terminadores, como lo son return, aserciones, saltos de linea, etc. pero el más importante para nosotros son las llamadas a funciones ya que estas permiten son las que nos permiten realizar análisis interprocedurales y son el principal medio de cambio entre bloques básicos. Aquí podemos obtener información de los parámetros formales y hacer análisis del perfil de la función, y también es donde realizamos las preparaciones para hacer que la información que tenemos almacenada sobre los análisis que realizamos sea consistente entre diferentes bloques con diferentes designaciones a una misma variable.

Una vez visitados todos los bloques básicos involucrados con la función, retornamos al archivo *analyzer.rs* donde procedemos a realizar los estudios de los datos obtenidos post-análisis. Aquí realizaremos el análisis de aliasing del grafo points-to y lo compararemos con los lifetimes obtenidos de Stacked Borrows para brindar información de la posible existencia de alias. Además, utilizando el paquete *dot* procederemos a imprimir el grafo en formato .dot para que sea más sencillo visualizarlo.

4.2.1. Análisis estáticos

Los principales análisis que realiza la herramienta son de Stacked Borrows y Points-to. Stacked borrows esta implementado dentro del archivo *stacked_borrows.rs* de una manera simple, consta de una pila la cual se va actualizando a medida de que nuestro interprete detecta creaciones y usos de los distintos tipos de variables y

Algorithm 1: Algoritmo de visita o recorrido del MIR

Input: Optimized MIR (CFG)

Function `VisitBody(`*body*`):`
- Get function information
- **foreach** *variable* ***in*** *body.local_declarations* **do**
  - Get variable debug info
- **foreach** *block* ***in*** *body.basic_blocks* **do**
  - VisitBlock(*block*)

Function `VisitBlock(`*block*`):`
- **foreach** *statement* ***in*** *block.statements* **do**
  - VisitStatement(*statement*
- VisitTerminator(*block.terminator*)

Function `VisitStatement(`*stmt*`):`
- **if** *stmt* ***is*** *an Assignment* **then**
  - **switch** *stmt.assign* **do**
    - **case** *Use* `/* Creation or mutation of a variable */`
      do Update state `/* Data used for analysis (Stacked Borrows or points-to) */`
    - **case** *Ref* `/* Use of a mutable or immutable reference */`
      do Update state
    - **case** *Cast* **do**
      - CastBehaviourCheck()
      - Update state
    - ...
  - ...

Function `VisitTerminator(`*term*`):`
- **if** *term* ***is*** *a Function call* **then**
  - $fcall \leftarrow term.call$
  - Get function info;
  - **foreach** arg ***in*** $fcall.arguments$ **do**
    - VisitArgument(arg)
  - CheckFunctionProfile()
  - Do preparation for inteprocedural analysis
  - VisitBody($fcall.body$) `/* Visit the function */`
  - Update state
- ...

referencias. A su vez, llevamos un vector con los nombres reales de las variables para que sea más fácil identificarlas en vez de usar la representación interna. La pila esta implementada utilizando el paquete **VecDeque** el cual se encuentra en la librería estándar de Rust, garantizando asi la velocidad y seguridad. Además de implementar las reglas del modelo explicadas en 3.2.2, agregamos una función adicional muy simple llamada *is_live(tag)* la cual corrobora si una variable se encuentra en la pila; y por lo tanto, la variable se encuentra viva.

De forma similar, en el archivo *points_to.rs* se encuentra la implementación del análisis de alias points-to. Para la implementación del grafo, utilizamos el paquete *petgraph::Graph* que si bien no es parte de la librería estándar, es una de las librerías de grafos más usadas y posee una documentación ampliamente detallada. Al igual, que en Stacked Borrows cada vez que se produce una asignación, corroboramos si se crea o modifica una variable para realizar cambios en el grafo. En el caso de que se cree una variable o referencia creamos un nodo y modificamos las aristas en caso de ser necesario. Cuando termina la ejecución del programa y se recorrieron todos los bloques y sentencias, procedemos a realizar el chequeo de si existen variables que son alias. Esto lo realizamos buscando todos los nodos que tengan más aristas entrantes que salientes; es decir, que existan más de 1 camino por el cual se pueda acceder a un nodo, dentro de la función (aliasing_test()) la cual retorna la lista de variables que pueden contener alias. Una vez obtenida esta lista, chequeamos si algunas de estas no están vivas con la información de Stacked Borrows e informamos todo por pantalla.

Durante el estudio de los terminadores, precisamente dentro de las llamadas a funciones en el archivo *MirVisitor/terminator_visitor.rs*, realizamos el análisis del perfil de la función a invocar. Este consiste en corroborar que no exista alias dentro de los parámetros de la misma, ya que si existiera podría indicar un potencial error de diseño de la función, especialmente si ambos son referencias mutables. Para hacer este análisis, tomamos los pares de argumentos y utilizando el grafo de points-to, verificamos si existe algún camino que conecte ambos nodos, dentro de la función *are_alias(a,b)*. En caso de que exista aliasing dentro de los parámetros de la función, se procede a comprobar si alguna ambas referencias son mutables y se muestra un mensaje en pantalla informando de la situación.

Algorithm 2: Algoritmo de chequeo si dos variables en particular son alias

Output: *true* if are alias
Function `AreAlias(`a`,`b`):`
    if *a or b not exists* **then return** *false*

    $DFS_a \leftarrow graph.dfs(a)$ `/* Get all nodes connected to` a `using Deep First Search */`
    $DFS_b \leftarrow graph.dfs(b)$ `/* Get all nodes connected to` b `using Deep First Search */`

    foreach *node* ***in*** DFS_a **do**
        if DFS_b*.contains(node)* **then return** *true*
    return *false*

El último de los análisis que realizamos es el chequeo de cambios de representación durante el uso de casts en Unsafe.Su implementación la podemos encontrar en el archivo *MirVisitor/block_visitor.rs*, y ocurre durante el tipo especial de asignaciones que poseen el MIR distingue como Cast. Además de utilizar la información para la actualización de Stacked Borrows y points-to, se accede a los datos de la representación de las variables involucradas y se obtiene la cantidad de bytes que necesitan. Una vez obtenida las cantidades, mediante una consulta al MIR, se procede a compararlas y en caso de que exista una diferencia se muestra el mensaje por pantalla informando del posible error.

Capítulo 5

Resultados obtenidos y trabajos futuros

5.1. Casos de uso

// A continuación se analizan ejemplos de programas que la herramienta detecta algo asi. // Agregar conclusiones y trabajos futuros.

5.1.1. Stacked Borrows y Análisis del perfil de funciones

Nuestra herramienta utiliza Stacked Borrows y un análisis dedicado a los enunciados de las funciones para detectar errores de aliasing. Esto lo podemos apreciar en el siguiente ejemplo.

```
fn main() {
    let mut local = 5;
    let raw_pointer = &mut local as *mut i32;
    let result = unsafe { example1(&mut *raw_pointer, &mut *raw_pointer) };
    println!("{}", result); // Prints "13".
}

fn example1(x: &mut i32, y: &mut i32) -> i32 {
    *x = 42;
    *y = 13;
    return *x; // Has to read 42 , because x and y cannot alias !
}
```

Listing 5.1: Ejemplo1 mostrado en el paper Stacked Borrows (Jung, 2020)

Al comienzo de la función *main()* podemos apreciar la declaración de una variable mutable(local) y un puntero raw(raw_pointer) que hará referencia a la misma. Luego se invoca la función *example1*, la cual toma como argumentos formales 2 referencias mutables, utilizando la referencia al puntero raw en ambos argumentos. Aquí es donde surgen los problemas. Primero, si bien no esta contemplado por Stacked Borrows, al llamar una función con más de una referencia mutable indica que la implementación del código puede no ser correcta. Y segundo, el problema que trata de atacar Stacked Borrows, es el grave problema de aliasing que surge cuando ambos argumentos tratan de modificar los valores de sus referencias, cuando están apuntando a una misma variable.

Para resolver el primer problema, nuestra herramienta analiza el perfil de las funciones que son llamadas. Mediante este análisis comprobamos que no existan dos o más referencias mutables apuntando a una misma variable a la hora de la invocación. Si los argumentos formales incluyen dos o más referencias mutables se muestra un mensaje "Caution" ya que puede llegar a generar problemas si no se toman las medidas de seguridad necesarias para la comprobación de los parámetros. Y en el caso de que existan dos o más referencias mutables que hacen referencia a una misma variable, es decir que exista aliasing, se muestra un mensaje de "Warning" porque es un gran indicativo de que pueden generarse errores debido a esto. Esto lo podemos apreciar en terminator_visitor.rs entre las lineas 33-50.

El segundo problema se resuelve mediante la implementación de Stacked Borrows. Generamos una pila donde iremos agregando las variables y referencias a medida que se van creando, y mortificándola cada vez que se produce una modificación a las mismas. Mediante esta pila simularemos el trabajo del borrow checker para determinar el tiempo de vida de las variables, extendiéndolo de tal manera que el análisis soporte tanto programas safe como unsafe. De esta manera, cada vez que la pila se modifica se realiza un chequeo para comprobar si existen referencias mutables X e Y tales que se encuentren ordenadas "XYXY" en la pila. Esto indicaría una violación de los principios de Stacked Borrows, y por lo tanto, se muestra un "Error". Estos errores pueden ser que la variable no tenia permisos para escribir o leer, dependiendo del caso. En el ejemplo, en la función *example1()* podemos ver como romper el principio de la pila genera errores. Queremos retornar 42 en la variable X, pero como existe aliasing y se realiza una escritura de la forma "XYX", el valor a retornar es diferente. Por lo tanto existe un error ya que la variable Y no debería haber tenido permitido escribir. Una posible solución seria cambiar los accesos para que queden de forma "YXXY", de esta manera no se rompen los principios establecidos y el programa seria considerado correcto.

En el siguiente fragmento de la salida de nuestra herramienta podemos apreciar estos mensajes indicativos dentro de la información del MIR.

```
Main body -- Start
Block Main bb0 --Start

use bb0[0] Assign local = const 5_i32
bb0[1] Assign 3 = &mut _1 ref local
bb0[2] Assign raw_pointer = &raw mut (*_3) ref 3
bb0[3] Assign 6 = &mut (*_2) ref raw_pointer
bb0[4] Assign 5 = &mut (*_6) ref 6
bb0[5] Assign 8 = &mut (*_2) ref raw_pointer
bb0[6] Assign 7 = &mut (*_8) ref 8
bb0[7] Terminator _4 = example1(move _5, move _7) -> bb1
        where _4 is result
        and _5 is 5     and _7 is 7
call example1
Caution: This function call contains two or more mutable arguments
WARNING: Calling function with two mutable arguments that are alias
const ty for<'r, 's> fn(&'r mut i32, &'s mut i32) -> i32 {example1}

Example1 body -- Start

Block Example1 bb0 --Start

use bb0[0] Assign x = const 42_i32
use ERROR Tag "y" does not have WRITE access
bb0[1] Assign y = const 13_i32
use ERROR Tag "x" does not have READ access
bb0[2] Assign 0 = (*_1) ref x
bb0[3] Terminator return

Block Example1 bb0 --End
Example1 body -- End
```

5.1.2. Análisis de cast

Mediante el uso de unsafe podemos realizar casts de un tipo determinado a otro diferente, el cual ni siquiera tenga la misma representación o al menos use una representación que tenga una cantidad de bytes equivalente. Esta utilización puede resultar en comportamiento indefinido. Un ejemplo de este mecanismo lo podemos encontrar en el ejemplo:

```
pub fn main() {
    let mut one: u64 = 5;
    let raw = &mut one as *mut u64;
```

```
    let raw2 = raw as *mut u32;
    unsafe {
        let two = *raw2;
    }
}
```

Listing 5.2: Ejemplo de casteo de una variable a otra con diferente representación.

En este ejemplo podemos apreciar como declaramos una variable llamada *one*, que es del tipo u64 y tiene una representación en 8 bytes. Luego creamos dos punteros raw, donde el primero hace referencia a la primer variable manteniendo su representación de u64. Sin embargo, en el segundo puntero (raw2), se crea a partir de raw y utilizando como base el tipo u32, que tiene una representación de 4 bytes y es diferente. En la siguiente instrucción dentro del bloque unsafe hacemos uso del valor dentro de raw2 colocandolo dentro de una variable nueva llamada *two*.
Esto es un problema grave, ya que la nueva variable *two* tiene dentro información que probablemente no sea correcta, ya que hicimos un casteo que no debería ser posible si utilizáramos unicamente referencias dentro de safe Rust. Estamos realizando una transformación de una variable u64 a una u32 que tiene una representación no solo diferente, sino que 4bytes menor, de una manera insegura y el contenido probablemente sea inconsistente y produzca comportamiento indeseado en el programa.

Nuestra herramienta detecta estas situaciones mediante la comparación del tamaño de la representación de ambas partes de un cast. Si el tamaño es igual o mayor no hay necesidad de mostrar ningún mensaje de error. Pero cuando queremos hacer una transformación a una representación menor, es muy probable que suframos de perdida de información; y por este motivo nuestro analizador lo detecta y envía un mensaje informando de la situación. Esto lo podemos observar programado en el archivo block_visitor.rs entre las lineas 122-150

En el output a continuación se observa el mensaje de Warning asociado a este caso.

```
Main body -- Start
Block Main bb0 --Start

use bb0[0] Assign one = const 5_u64
bb0[1] Assign 3 = &mut _1 ref one
bb0[2] Assign raw = &raw mut (*_3) ref 3
use bb0[3] Assign 5 = _2 ref raw
kst WARNING: Casting from a layout with 8 bits to 4 bits
from *mut u64 to *mut u32
bb0[4] Assign raw2 = move _5 as *mut u32 (Misc) ref 5
use bb0[5] Assign two = (*_4) ref raw2
bb0[6] Terminator return

Block Main bb0 --End
Main body -- End
```

5.1.3. Points to alias analysis

Otro apartado importante en el cual se enfoca nuestro proyecto es en el análisis interprocedural de alias. Mediante el uso de points-to en conjunto con los lifetimes obtenidos gracias al stack de Stacked Borrows, podemos obtener un grafo de aliasing que nos brinda información adicional para detectar errores.

```
fn main() {
    let mut num = 5;

    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;

    unsafe {
        let res = *r1 + *r2;
    }
```

```
}
```

Listing 5.3: Ejemplo de un programa el cual le aplicamos analisis de alias points-to (Hind and Burke, 1999)

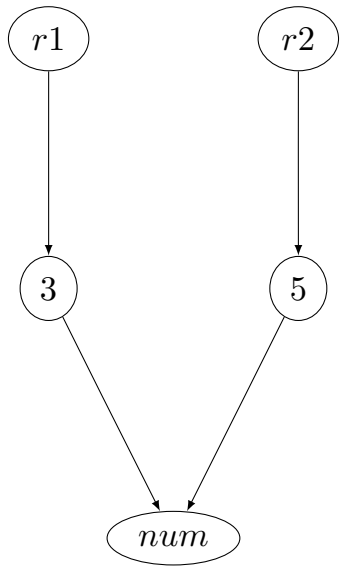
En este fragmento de código podemos ver como se declara una variable mutable *num*, y se crean dos punteros raw que hacen referencia a esta (*r1* y *r2*). Luego se genera una variable *res* que contendrá el valor resultante de la suma de los valores apuntados por *r1* y *r2*, que al ser raw pointers, para ser derreferenciados deben estar dentro de un bloque unsafe.

En este ejemplo, Stacked Borrows no genera ningún error o advertencia, ya que el programa es válido. Sin embargo, sigue existiendo aliasing y esto puede representar un problema a futuro si se llegase a modificar el código. Ahi es donde entra nuestro points-to análisis, que genera un grafo de alias a medida que se construye el programa y una vez terminado, lo observa para buscar potenciales problemas de alias. Si encuentra alguno, lo reporta por pantalla al finalizar el análisis de cada una de las funciones. Además, gracias a Stacked Borrows podemos saber si esa misma variable esta viva o muerta, y en caso de que se encuentre en este ultimo estado, también lo mencionamos por pantalla.

En la salida de la ejecución de nuestra herramienta podemos observar como nos advierte del posible problema de aliasing de *num* y mediante la vista una parte del grafo (utiliza formato DOT) vemos cuales son las variables definidas por el usuario y auxiliares que apuntan a *num*

```
Block Main bb1 --End
Main body -- End

Variable num may have aliasing
digraph {
    0 [ label = "num"]
    1 [ label = "3"]
    2 [ label = "r1"]
    3 [ label = "5"]
    4 [ label = "r2"]
    5 [ label = "7"]
    6 [ label = "8"]
    7 [ label = "9"]
    8 [ label = "_res"]
    1 -> 0 [ ]
    2 -> 1 [ ]
    3 -> 0 [ ]
    4 -> 3 [ ]
    8 -> 7 [ ]
}
```



Bibliografía

Aldrich, J. (2017). Pointer analysis.

Astrauskas, V., Matheja, C., Poli, F., Müller, P., and Summers, A. J. (2020). How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27.

Beingessner, A. (2021). Learning rust with entirely too many linked lists.

Beingessner, A. and Klabnik, S. (2016). The rustonomicon: The dark arts of advanced and unsafe rust programming.

Hind, M. and Burke, M. (1999). Interprocedural pointer alias analysis. *Proceedings of the ACM on Programming Languages*, 21(4):848–894.

Jung, R. (2020). Stacked borrows: An aliasing model for rust. *Proceedings of the ACM on Programming Languages*, 4(41).

Klabnik, S. and Nichols, C. (2019). *The Rust Programming Language*. No strach press.

Li, Z., Wang, J., Sun, M., and Lui, J. C. (2021). Mirchecker: detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2183–2196.

M, M. (2022). Guide to rustc development. Distribuido en https://rustc-dev-guide.rust-lang.org/.

Olson, S. (2016). Miri: An interpreter for rust's mid-level intermediate representation. Master's thesis, University of Saskatchewan.

Presman, R. (1997). *Ingeniería del software: un enfoque práctico*. McGRAW-HILL, 7 edition.

Shapiro, M. and Horwitz, S. (1997). Fast and accurate flow insensitive points to analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY. ACM.

TheRustCommunity (2021). Rust by example.

Yechan and Youngsuk (2021). Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, New York, NY. ACM.