Análisis estático extendido de código Rust

Ezequiel Giachero

6 de diciembre de 2022

Índice general

Capítulo 1

Introducción

1.1. Que es Rust? Cuál es nuestro objetivo?

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.

A lo largo de la tesis explicaremos las particularidades del lenguaje y su compilador, para luego poder explicar como diseñamos y desarrollamos esta herramienta, finalizando con muestras de su uso y hablado del futuro del mismo. En el resto del apartado introductorio hablaremos más de los conceptos básicos de Rust, los sistemas de *ownership* y *borrow*, ejemplos de la necesidad del unsafe rust y comenzaremos a introducir el sistema de tipos de Rust. 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. 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.

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, 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. 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*.

Capítulo 2

El sistema de tipos de Rust

Capítulo 3

Extendiendo el borrow checker

Capítulo 4

La herramienta: diseño e implementación

Capítulo 5

Resultados obtenidos y trabajos futuros

5.1. Casos de uso

5.1.1. Stacked Borrows y Análisis de cabecera 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 la cabecera 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)

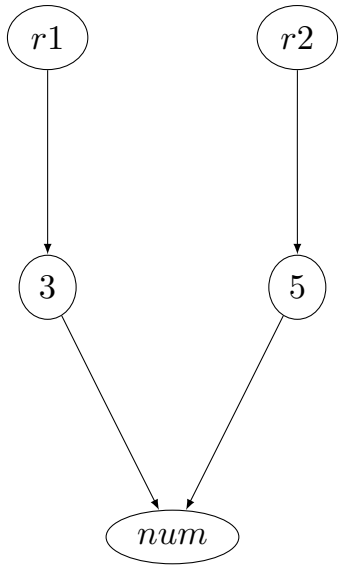
En este fragmento de código podemos ver como se declara una variable mutable *num*, y se crean dos punteros raw que hacen referencia a esta (*r1* y *r2*). Luego se genera una variable *res* que contendrá el valor resultante de la suma de los valores apuntados por *r1* y *r2*, que al ser raw pointers,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

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. 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.

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.