Rust Static Analyzer

Ezequiel Giachero

25 de noviembre de 2022

Índice general

Capítulo 1

Introducción

1.1. Que es 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. Tal 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, ademas de ser muy rápido y eficiente ya que 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 dia Rust es considerado como una alternativa (inclusive más segura) a C y C++

Sin embargo, al colocar tantas restricciones para mantener la seguridad de todos los programas escritos en Rust, resulta complejo o incluso imposible crear ciertas estructuras o programas respetando todos las comprobaciones de seguridad que el lenguaje por defecto implementa. Debido a esto, surge la necesidad del código unsafe en Rust, en donde se deshabilitan ciertas de estas restricciones y queda la responsabilidad del programador crear programas seguros y sin fallos.

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 en proyectos como MIRI o Rudra, que contienen una gran cantidad de herramientas para detectar errores específicos en código safe o 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.

El presente trabajo tiene como objetivo crear una herramienta específica 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, enfocarnos en encontrar problemas de aliasing, o mediante Stacked Borrows problemas de referencias colgadas que el borrow-checker se habría encargado.

A lo largo de la tesis explicaremos las particularidades del lenguaje y su compilador, para luego poder explicar como diseñamos y desarrollamos esta herramienta. En el resto del capitulo 1 hablaremos un poco mas sobre el lenguaje safe y la necesidad del unsafe, junto con las características principales de Rust como lo son el sistema de *ownership* y *borrow*. En el capitulo 2 nos centraremos en hablar específicamente del sistema de tipos de rust centrándonos en el ámbito unsafe. Luego, en el capitulo 3, hablaremos sobre Stacked Borrows (Jung, 2020) y diferentes análisis estáticos realizados, y conceptualmente como hicimos para extender el *borrow-checker*. El cuarto capitulo nos centraremos en como diseñamos e implementamos la herramienta, para luego en el capitulo 5 mostrar ejemplos del funcionamiento de la misma y observar cual podría ser el futuro del proyecto.

Capítulo 2

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

```
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

```
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

```
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.

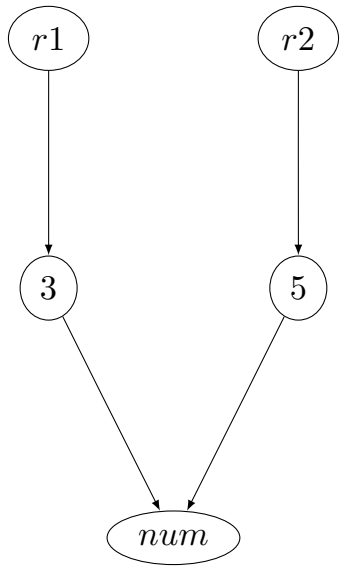
En este ejemplo, Stacked Borrows no genera ningún error o advertencia, ya que el programa es válido. Sin embargo, sigue existiendo aliasing y esto puede representar un problema a futuro si se llegase a modificar el código. Ahi es donde entra nuestro points-to análisis, que genera un grafo de alias a 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

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.

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.