Rust Static Analyzer

Ezequiel Giachero

21 de noviembre de 2022

Índice general

Capítulo 1

Introducción

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

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 [1]

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 Çautionza 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 ."Er"ror". 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 example(arg: &mut u32) -> u32 {
    let local = arg;
    *local = 15;
    1
}

fn main() {
    let mut id = 5;
    let arg = &mut id;
    let _pointer = &arg;
    let _result = example(arg);
}
```

Bibliografía

[1] RALF JUNG. *Stacked Borrows: An Aliasing Model for Rust.* 2020.

[2] CAROL NICHOLS STEVE KLABNIK. *The Rust Programming Language.* 2021.

[3] MARK M. *Guide to Rustc development.* 2022.

[4] ROGER PRESMAN. *Ingeniería del software: un enfoque práctico.* 1997.

[5] YOUNGSUK YECHAN. *RUDRA: Finding Memory Safety Bugs in Rust at the Ecosystem Scale.* 2021.

[6] SCOTT OLSON. *Miri: An interpreter for Rust's mid-level intermediate representation.* 2016.

[7] MICHAEL BURKE MICHAEL HIND. *Interprocedural Pointer Alias Analysis.* 2019.

[8] SUSAN HORWITZ MARC SHAPIRO. *Fast and Accurate Flow Insensitive Points To Analysis.* 1997.