Rust Safeness

Edoardo De Matteis

14 giugno 2020

Indice

Capitolo 1

Introduzione

Il linguaggio di programmazione *Rust* è noto e apprezzato principalmente per la sua efficienza computazionale e per le sue garanzie sulla sicurezza [6, 1], a tal proposito è molto difficile parlare di sicurezza in generale dato che non ne esiste una definizione assoluta, è un concetto relativo che varia in base a cosa ci interessa proteggere e garantire. La sicurezza di un'organizzazione - che essa sia una multinazionale o un singolo individuo - è definita da una **security policy**: un documento contenente regole, principi e pratiche che determinano come garantire che il sistema si trovi in uno stato sicuro [5]. Nell'ambito della sicurezza informatica con l'avvento di internet e la sempre più grande condivisione di codice è nata la necessità di verificare la sicurezza di programmi e per estensione dei linguaggi di programmazione, per dimostrare delle proprietà di un linguaggio è necessario avere una definizione formale della sua **semantica** ovvero l'insieme delle regole che definiscono il significato di un linguaggio e dei suoi programmi.
In *Rust* un programma è sicuro se rispetta **memory safety** (capitolo 2) e **semantica statica** (capitolo 3) [8, 14] ma la documentazione presenta delle definizioni poco chiare se non addirittura contraddittorie:

> "Rust code is incorrect if it exhibits any of the behaviors in the following list. This includes code within `unsafe` blocks and `unsafe` functions. `unsafe` only means that avoiding undefined behavior is on the programmer; it does not change anything about the fact that Rust programs must never cause undefined behavior. [...] There is no formal model of Rust's semantics for what is and is not allowed in unsafe code" [8, 14.3]

L'uso del termine "unsafe" è causa di confusione dal momento che `unsafe` è un costrutto sintattico che consente la scrittura di codice che può causare **undefined behavior**, per questo si dice che Rust è composto da due "sotto-linguaggi": *unsafe Rust* e *safe Rust* in base alla presenza o meno di un blocco `unsafe`, questo costrutto è desiderabile perché permette sia di poter comunicare con librerie in *C* che di avere un accesso più esplicito alla memoria [6, 1.2]. Piuttosto che una definizione formale di undefined behavior si ha una lista in

continuo aggiornamento [8, 14.3]. Nella citazione riportata sono presenti fonti di confusione: nella stessa frase viene detto prima che in un blocco `unsafe` sia compito del programmatore evitare undefined behavior e subito dopo viene affermato come le garanzie sulla safety siano le stesse di un blocco `safe`, quindi anche l'assenza di undefined behavior. Volendo comunque considerare solo l'ultima affermazione, l'esecuzione in un blocco `unsafe` di uno degli esempi in lista non dovrebbe causare undefined behavior; nel listato 1.1 si ha un **raw pointer dereferencing** (presente in lista come dimostrato in figura 1): dato un puntatore riesco a leggere il dato memorizzato nella locazione puntata tramite l'aritmetica dei puntatori tipica del C. È ragionevole che in unsafe Rust questo sia possibile ma mette in luce come la documentazione sia poco chiara.

```
fn main() {
    unsafe {
        let x = 42;
        let ptr = &x as *const i32;
        println!("Dereferencing: {}", *ptr);
    }
}
```

```
$ rustc raw.rs -o bin/raw
$ ./bin/raw
Dereferencing: 42
```

Listing 1.1: Raw pointer dereferencing in un blocco `unsafe`

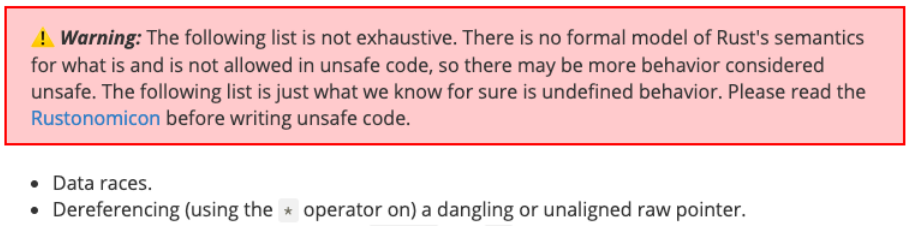

Figura 1.1: Il raw pointer dereferencing è considerato undefined behavior

Capitolo 2

Memory Safety

2.1 Configurazione della memoria

Consideriamo tre diversi tipi di allocazione della memoria presenti nei linguaggi C e Rust:

- **Allocazione statica**. Il compilatore memorizza solo ed esclusivamente le variabili globali, il nome deriva dal fatto che in C una variabile in una funzione è dichiarata globalmente tramite la keywork `static`.
- **Allocazione automatica**. Se durante l'esecuzione del codice è possibile sapere in anticipo quanta memoria dovrà essere allocata si esegue allocazione automatica tramite una struttura dati detta **stack** o **call stack**, la cui particolarità è quella di crescere verso il basso; alcune entità che richiedono allocazione automatica sono le variabili di tipo primitivo e le funzioni. Sia con l'allocazione statica che automatica la memoria da allocare è nota già a tempo di compilazione, la sequenza delle operazioni però è nota solo durante l'esecuzione e questo giustifica la necessità di questo secondo tipo di allocazione.
- **Allocazione dinamica**. Per oggetti la cui dimensione non è prevedibile (come tipi composti o liste dinamiche) si usa una struttura dati nota come **heap** che ha la particolarità di crescere verso l'alto. Proprio a causa del fatto che heap e stack crescono in direzioni opposte si può causare un **overflow** (sezione 2.2.1). Ad ogni modo quando si chiama una funzione che usa una struttura dinamica nello stack viene salvato il puntatore di tale struttura esplicitamente in linguaggi come il C, all'insaputa del programmatore in *Java* ed esplicitamente in Rust ma con dei controlli che evitano comportamenti non sicuri.

Questo approccio è stato reso popolare dal C ed è oggi adottato da numerosi linguaggi, sempre in C esiste un ulteriore tipo di allocazione della memoria noto come **register allocation** che permette di scrivere direttamente su un blocco

del processore; non essendo presente in Rust non viene approfondita.

2.2 Violazione della memory safety

Definiamo un'entità software **memory safe** quando non accede ad indirizzi di memoria fuori dal proprio **address space** (una zona di memoria dedicata all'entità software sopra citata) né esegue istruzioni fuori dall'area assegnatale da compilatore e linker [2], un linguaggio che garantisce la memory safety dei suoi programmi è detto memory safe.
I linguaggi in cui la gestione della memoria è lasciata al programmatore sono generalmente unsafe (come C o *C++*), altri linguaggi quali Java o *C#* risolvono il problema nascondendo i puntatori e gestendo automaticamente la memoria, obbligando il linguaggio ad eseguire i dovuti controlli e spesso usando un **garbage collector** che rende nuovamente disponibile la memoria allocata. In safe Rust è presente l'aritmetica dei puntatori e non si ha garbage collector, questo è possibile grazie ai meccanismi di **ownership** (sezione 2.2.4) e **borrowing** (sezione 2.2.5).

2.2.1 Buffer Overflow

Quando in un buffer si possono inserire dati di dimensione maggiore della sua capacità sovrascrivendo così altre informazioni [9] si può parlare di buffer overflow.
Un esempio esplicativo ed interessante di buffer overflow in C è nel listato 2.1 [7, 7.5]: dato che gli array sono posizionati uno dopo l'altro (in ordine LIFO essendo dati allocati su uno stack) si riesce a leggere e scrivere su `str1` eludendo anche il controllo di uguaglianza, nel terzo caso questo controllo viene superato perché si confrontano solo i primi 8 caratteri. In Rust questo non è possibile, si veda nel listato 2.2 come il linguaggio esegua dei controlli sulle dimensioni degli array, esattamente ciò che la funzione `gets` non fa.

```c
#include <stdio.h>
#include <string.h>

int main(int argc,char* argv[]) {
  int valid = 0;
  char str1[8];
  char str2[8];

  strcpy(str1, "START");
  gets(str2);
  if(strncmp(str1, str2, 8) == 0) valid = 1;
  printf("str1:%s, str2:%s, valid:%d\n", str1, str2, valid);
}
```

```
$ ./bin/stackattack_c
warning: this program uses gets(), which is unsafe.
START
```

```
str1:START, str2:START, valid:1

$ ./bin/stackattack_c
warning: this program uses gets(), which is unsafe.
EVILINPUTVALUE
str1:TVALUE, str2:EVILINPUTVALUE, valid:0

$ ./bin/stackattack_c
warning: this program uses gets(), which is unsafe.
BADINPUTBADINPUT
str1:BADINPUT, str2:BADINPUTBADINPUT, valid:1
```

Listing 2.1: Stack overflow in C

```
fn main() {
    let arr: [i32; 2] = [13, 10];
    arr[4];
}
```

```
$ rustc bufof.rs -o ./bin/bufof_rs
error: index out of bounds: the len is 2 but the index is 4
 --> bufof.rs:3:5
  |
3 |     arr[4];
  |     ^^^^^^
  |
  = note: `#[deny(const_err)]` on by default

error: aborting due to previous error
```

Listing 2.2: Buffer overflow in Rust

2.2.2 Stack Overflow

Quando si chiama una funzione si salva lo **stack frame** (noto anche come **activation record** è una zona del call stack contenente dati necessari alla funzione attualmente in esecuzione) e può capitare che chiamate ricorsive facciano entrare il programma in uno stato di non terminazione, Rust individua a tempo di compilazione la ricorsione infinita e segnala l'overflow a tempo di esecuzione. In C l'errore che si vede è dato dal sistema operativo (macOS Catalina 10.15.3) perché con questa ricorsione infinita si prova ad accedere ad un frammento di memoria protetto.

```
void f() {f();}

int main() {
  f();
}
```

```
$ ./bin/segfault_c
[1]    22217 segmentation fault  ./bin/segfault_c
```

Listing 2.3: Stack overflow ricorsivo in C

```rust
fn main() { f();}

fn f() {
    f();
}
```

```
$ rustc segfault.rs -o bin/segfault_rs
warning: function cannot return without recursing
 --> segfault.rs:3:1
  |
3 | fn f() {
  | ^^^^^^ cannot return without recursing
4 |     f();
  |     --- recursive call site
  |
  = note: `#[warn(unconditional_recursion)]` on by default
  = help: a `loop` may express intention better if this is on
    purpose

$ ./bin/segfault_rs

thread 'main' has overflowed its stack
fatal runtime error: stack overflow
```

Listing 2.4: Stack overflow ricorsivo in Rust

2.2.3 Integer Overflow

In qualsiasi macchina non astratta si ha memoria finita ed è rappresentabile solo un insieme finito di numeri, quando un valore è troppo grande (o piccolo) per essere rappresentato si ha un overflow. Sono due gli approcci principali per risolvere un integer overflow, ognuno basato su un'aritmetica differente:

- **Modular arithmetic**. Si applica un wrapping ad ogni numero in overflow, dato il valore n e una memoria a m bit si memorizza $w = n \mod m$.
- **Saturation arithmetic**. Dati min e max - rispettivamente il numero più grande e più piccolo rappresentabile - e un numero n in memoria viene salvato

$$c = \begin{cases} max & n > max \\ min & n < min \\ n & \text{altrimenti} \end{cases}$$

L'integer overflow ha in passato causato problemi molto seri: durante il volo inaugurale del lanciatore *Ariane 5* un integer overflow dovuto ad una conversione in intero a 16 bit di un float a 64 bit ha causato una reazione a catena per cui il razzo ha virato orizzontalmente distruggendosi poco dopo il lancio, il codice scritto in Ada non prevedeva controlli di overflow come esplicitamente richiesto dai progettisti per motivi di efficienza. In Rust si hanno due modalità di compilazione: in **debug mode** si eseguono dei controlli in più rispetto alla **release mode** tra cui controlli dinamici per l'integer overflow - listato 2.5 -

contrariamente al C che applica direttamente wrapping.
Differenti linguaggi adottano differenti approcci: se si può prevedere il valore massimo possibile in un programma allora è in generale facile evitare integer overflow; in caso contrario si possono utilizzare metodi dinamici come in Rust.

```
fn main() {
    let mut n = 1;
    let mut i = 1;

    while n > 0 {
        n = n*2;
        i = i+1;
        println!("{} bit architecture.", i);
    }
}
```

```
$  ./bin/intof_rs
2 bit architecture.
3 bit architecture.
        ...
30 bit architecture.
31 bit architecture.
thread 'main' panicked at 'attempt to multiply with overflow',
    intof.rs:6:13
note: run with `RUST_BACKTRACE=1` environment variable to display a
     backtrace.
```

Listing 2.5: Integer overflow in Rust

Nel listato 2.6 si prende un intero come primo argomento e una stringa - idealmente di lunghezza pari al primo argomento - come secondo. Il problema sorge nella conversione da integer a short, inserendo come primo argomento 65536 (2^{16} che non è rappresentabile con i soli 16 bit di uno short) si causa un buffer overflow, combinandone la pericolosità con la difficoltà di rivelamento dell'integer overflow.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3) return -1;
    i = atoi(argv[1]);
    s = i;

    if(s >= 80) return -1;

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);
```

```
21
22      return 0;
23  }
```

```
1  $ ./bin/width1 65536 hello
2  s = 0
3  [1]    20475 illegal hardware instruction  ./bin/width1 65536 hello
```

Listing 2.6: Integer overflow in C

2.2.4 Double-Free

Si ha un double free error [3, 10.4.4] quando si prova a liberare più volte la stessa zona di memoria e in Rust non è possibile grazie all'ownership, questa rappresenta il possesso di un'area memoria e impedisce a questa di esser puntata da più puntatori, nel listato 2.7 si può vedere come `s1` perda validità dopo aver trasferito a `s2` l'ownership della stringa cui puntava.

```
1  fn main() {
2      let s1 = String::from("Primo!");
3      let _s2 = s1;
4      s1;
5  }
```

```
1  $ rustc doublefree.rs -o bin/doublefree_rs
2  error[E0382]: use of moved value: ‘s1‘
3   --> doublefree.rs:4:5
4    |
5  2 |      let s1 = String::from("Primo!");
6    |          -- move occurs because ‘s1‘ has type ‘std::string::
     String‘, which does not implement the ‘Copy‘ trait
7  3 |      let _s2 = s1;
8    |                -- value moved here
9  4 |      s1;
10   |      ^^ value used here after move
11
12 error: aborting due to previous error
13
14 For more information about this error, try ‘rustc --explain E0382‘.
```

Listing 2.7: Ownership

In C invece si può osservare il codice nel listato 2.8: si immagini di avere un servizio ad iscrizione salvando ogni utente tramite una struct `User`: Guido libera la propria memoria che, disponibile, potrà memorizzare `Luisa` che si è appena iscritta. Il puntatore `Guido` però ora punta all'account di Luisa e con `free(Guido)` proprio l'account di Luisa viene eliminato con il risultato che non solo Luisa non ha più un account ma a quello di Carla possono accedere sia Guido che Luisa.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
```

```c
typedef struct User User;

struct User {
    User * self;
    char name[8];
};

int main(int argc, char** argv) {

    // (1) Guido si iscrive
    struct User * Guido = malloc(sizeof(struct User));
    Guido->self = Guido;
    strcpy(Guido->name, "Guido");

    printf(
        "Guido: [self: %p, name: %s]\n",
        Guido->self, Guido->name
    );

    // (2) Guido libera la sua memoria
    free(Guido);

    // (3) Luisa si iscrive
    struct User * Luisa = malloc(sizeof(struct User));
    Luisa->self = Luisa;
    strcpy(Luisa->name, "Luisa");

    printf(
        "Luisa: [self: %p, name: %s]\n",
        Luisa->self, Luisa->name
    );

    // (4) Guido libera la memoria di Luisa
    free(Guido);

    // (5) Carla si iscrive sulla memoria di Luisa
    struct User * Carla = malloc(sizeof(struct User));
    Carla->self = Carla;
    strcpy(Carla->name, "Carla");

    printf(
        "Carla: [self: %p, name: %s]\n",
        Carla->self, Carla->name
    );
    printf(
        "Luisa: [self: %p, name: %s]\n",
        Luisa->self, Luisa->name
    );
}
```

```
./bin/doublefree_c
Guido: [self: 0x7f9f72c05a30, name: Guido]
Luisa: [self: 0x7f9f72c05a30, name: Luisa]
Carla: [self: 0x7f9f72c05a30, name: Carla]
Luisa: [self: 0x7f9f72c05a30, name: Carla]
```

Listing 2.8: Double free in C

Il puntatore `Guido` è anche un **dangling pointer**.

2.2.5 Dangling References

Quando un oggetto viene eliminato ma il suo puntatore no si ha un dangling pointer che permettere di accedere a memoria cui non dovrebbe.
Immaginiamo ora che per vendicarsi Luisa abbia scritto un semplice sistema di messaggistica per leggere la corrispondenza di Guido, la procedura `send_message` crea un puntatore di tipo `Message` che ha scope solo ed esclusivamente dentro `send_message` ma facendo riferimento esplicito a quella locazione Luisa riesce a leggere il messaggio di Guido anche dopo che questo non esiste più. Nel listato 2.9 l'indirizzo viene stampato a schermo e inserito dall'utente (indicato da ">") per semplicità, lo si può rendere in modo più esplicito mantenendo una variabile per il puntatore, sia essa `mess` stessa o una ad hoc passata come argomento alla funzione.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Message {
  char from[8];
  char to[8];
  char text[32];
};

void send_message(char *, char *, char *);

int main() {
  send_message("Guido", "Carla", "asa nisi masa");

  char * ptr;
  printf("> ");
  scanf("%p", &ptr);

  printf("[%s, %s, %s]\n", ptr, ptr+8, ptr+16);
}

void send_message(char * from, char * to, char * text) {
  struct Message * mess = malloc(sizeof(struct Message));
  strcpy(mess->from, from);
  strcpy(mess->to, to);
  strcpy(mess->text, text);
  printf("%p\n", mess);
}
```

```
$ ./bin/dangref_c
0x7fec9fe04d10
> 0x7fec9fe04d10
[Guido, Carla, asa nisi masa]
```

Listing 2.9: Dangling pointer in C

In Rust non possiamo usare i puntatori perché l'ownership ci obbliga ad avere un puntatore solo e una volta disabilitato non possiamo più accedere a

quell'area di memoria né si può leggere la locazione perché non è permesso il raw dereferencing; esistonono però le reference [4, 4.2] che permettono di prendere in prestito (borrowing) una variabile senza esserne proprietari, nel listato 2.10 `s` nella riga 5 è quindi una **shallow copy** di `s2`, se una **deep copy** è la copia di tutto il contenuto di un oggetto con una shallow copy si memorizza solo il puntatore. Come con l'ownership anche nel borrowing si ha validità solo dentro il proprio scope.

```
1  fn main() {
2      let s1: &String;
3      {
4          let s2 = String::from("Hello");
5          s1 = &s2;
6      }
7      println!("{}", s1);
8  }
```

```
1  $ rustc dangref.rs -o bin/dangref_rs
2  error[E0597]: ‘s2‘ does not live long enough
3   --> dangref.rs:5:14
4    |
5  5 |          s1 = &s2;
6    |               ^^^ borrowed value does not live long enough
7  6 |      }
8    |      - ‘s2‘ dropped here while still borrowed
9  7 |      println!("{}", s1);
10   |                     -- borrow later used here
11
12 error: aborting due to previous error
13
14 For more information about this error, try ‘rustc --explain E0597‘.
```

Listing 2.10: Dangling reference in Rust

Capitolo 3

Type Safety

Ogni oggetto in matematica ha un **tipo**, basti pensare al fatto che non è possibile eseguire un'operazione come $\{13\} \wedge 10$ dato che $\{13\}$ è un insieme, 10 è un numero e $\wedge$ è un connettivo logico.
Lo stesso vale nei linguaggi di programmazione: l'insieme delle regole che assegna un tipo ad ogni espressione è detto semantica statica o **sistema dei tipi**; volendo dare una definizione di **type safety** si tenga conto che durante l'esecuzione di un programma si può incorrere in due tipi di errori: **trapped** e **untrapped**, i primi sono facilmente riconoscibili dato che interrompono l'esecuzione, i secondi no e per questo sono molto più difficili da individuare. Un linguaggio in cui nessun programma ben tipato - nei linguaggi con **tipizzazione statica** è onere del programmatore specificare il tipo di ogni variabile, si suppone quindi che non ci siano errori durante questa fase - genera errori trapped né untrapped si dice **sound**, solo untrapped invece si dice **safe**, un linguaggio sound è necessariamente safe [1].
Una definizione simile è quella di Milner per cui

well-typed programs cannot "go wrong" [5]

con programmi che si "comportano male" si possono intendere programmi non sound o che generano undefined behavior (sezione 1).

3.1 Violazione della type safety

3.1.1 Type Confusion

Nel listato 3.1 si alloca un array di short (2 B) staticamente, andrà quindi sullo stack e considerando che il computer su cui ho eseguito il programma monta un processore Intel - little endian- si ha la seguente rappresentazione in memoria

1	1000000000000000
0	0000000000000000

il puntatore `p` punta al primo elemento ma essendo `int` legge 4 B

00000000000000001000000000000000

che in decimale equivale a $2^{16} = 65536$.

```c
#include <stdio.h>

int main() {
  short arr[] = {0, 1};
  int* p = &arr[0];
  printf("%u", *p);
}
```

```
$ ./bin/lilliput_c
65536
```

Listing 3.1: Type confusion in C

In Rust non si può proprio puntare ad una variabile con un tipo differente da quello della variabile stessa.

```rust
fn main() {
    let arr: [i16; 2] = [0, 1];
    let p = &arr as *const i32;
}
```

```
$ rustc lilliput.rs -o bin/lilliput_rs
error[E0308]: mismatched types
 --> lilliput.rs:3:13
  |
3 |     let p = &arr as *const i32;
  |             ^^^^^^^^^^^^^^^^^^ expected i16, found i32

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
```

Listing 3.2: Type confusion in Rust

Capitolo 4

Note al Professore

Nella sezione 3.1.1 non sono riuscito a trovare esempi significativi di type confusion in C proprio grazie alla semplicità del linguaggio e assenza di classi quindi sembra più un'opacità semantica come ha detto Lei.

Bibliografia

[1] Pietro Cenciarelli. Dispense del corso linguaggi di programmazione, 2019.

[2] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, 2003.

[3] Dietere Gollmann. *Computer Security.* John Wiley & Sons Ltd, 3 edition, 2011.

[4] Steve Klabnik and Carol Nichols. The rust programming language, 2020.

[5] Robin Milner. A theory of type polimorphism in programming. *Journal of Computer and Sciences*, 7:348–375, 4 1978.

[6] Rustonomicon, 2020.

[7] William Stallings. *Operating Systems: Internals and Design Principles.* Pearson Education, 7 edition, 2012.

[8] The rust reference, 2020.

[9] Jansen Wayne, Theodore Winograd, and Karen Scarfone. Guidelines on active content and mobile code. Technical Report 800-28, NIST, 3 2008.