SAPIENZA
UNIVERSITÀ DI ROMA

Proprietà di sicurezza nel linguaggio Rust

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

Candidato

Edoardo De Matteis
Matricola 1746561

Relatore

Pietro Cenciarelli

Anno Accademico 2019/2020

Tesi non ancora discussa

Proprietà di sicurezza nel linguaggio Rust
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Versione: 10 novembre 2020

Email dell'autore: edoardodematteis@icloud.com

Indice

Capitolo 1

Introduzione

Il linguaggio di programmazione C è stato sviluppato nel 1978 e ad oggi è ancora uno dei più popolari [19] [13], ciò è dovuto al fatto che circa il 98% [2] dei sistemi informatici sul mercato siano sistemi integrati che richiedono un basso overhead e C risponde a questa esigenza tramite una gestione manuale della memoria, l'esecuzione dei programmi consuma quindi meno risorse ma si richiede un maggiore impegno da parte del programmatore, è infatti suo compito allocare memoria adeguatamente e successivamente liberarla. Rimettendo totalmente al programmatore la gestione della memoria C permette di scrivere programmi non sicuri, esistono linguaggi come Java considerati sicuri che gestiscono automaticamente la memoria tramite **garbage collection** (definizione 1.1) al prezzo di un maggiore overhead, sono quindi poco adatti alla scrittura di codice per embedded systems.

Garbage collection

Definizione 1.1. È una delle tecniche di gestione automatica della memoria più usate nei linguaggi di programmazione: tramite un modulo noto come **garbage collector** si tiene traccia di ogni allocazione e le zone di memoria non più necessarie vengono periodicamente liberate, rendendole nuovamente disponibili. La garbage collection ha il vantaggio di esonerare il programmatore dal dover gestire la memoria per potersi concentrare sulla logica del programma e la sua leggibilità, si evitano inoltre vulnerabilità quali double free o dangling pointer (rispettivamente sezioni 4.3 e 4.4). Con la garbage collection l'esecuzione dei programmi è più lenta data la presenza del garbage collector.

Nel breve futuro ci si aspetta sempre un numero crescente di oggetti d'uso comune connessi tra loro (i.e. internet of things) e si sente la necessità di metodi sicuri per programmare sistemi integrati. Nella ricerca di un linguaggio sicuro con gestione esplicita della memoria Rust sembra essere il miglior candidato grazie ai **lifetime**, questi descrivono lo scope nel quale una variabile è valida e sono fondamentali per il corretto funzionamento dei meccanismi di **ownership** e **borrowing**, dal momento che la loro verifica avviene a tempo di compilazione l'esecuzione dei programmi ne guadagna in velocità. I due obiettivi di Rust sono infatti sicurezza e velocità d'esecuzione dei suoi programmi: la prima è garantita dall'ownership che impedisce

il manifestarsi di **side effect** (definizione 1.2), la seconda sia tramite il maggior numero possibile di controlli statici che tramite borrowing il quale simula una call by reference - in Rust come in C c'è solo call by value.

Side effect

Definizione 1.2. Un programma presenta side effect quando contiene almeno un'operazione che modifica lo stato di una o più variabili non presenti nel suo ambiente locale. I side effect non sono necessariamente negativi, variabili globali e processi concorrenti acquistano la loro forza proprio da questi effetti collaterali.

Capitolo 2

Rust

2.1 Sistemi di tipi substrutturali

I sistemi di tipi **substrutturali** sono una famiglia di sistemi di tipi (capitolo 5) dove almeno una delle seguenti proprietà - dette **strutturali** - non è presente o lo è sotto determinate condizioni, allo stesso modo si parlerà anche di logiche substrutturali.

2.1.1 (Exchange) *L'ordine degli elementi all'interno di un'ipotesi o conclusione è irrilevante.*

2.1.2 (Weakening) *L'ipotesi e la conclusione possono essere estese con affermazioni non rilevanti senza alcuna ripercussione.*

2.1.3 (Contraction) *In ipotesi o conclusione è possibile sostituire due membri con un terzo, a patto che entrambi siano nella stessa espressione e che siano effettivamente unificabili.*

Un'interpretazione delle logiche e dei sistemi substrutturali permette di ragionare in termini di consumo di risorse, vincolandone l'accesso e tenendo traccia dei cambiamenti di stato della memoria evitando stati non validi.

In un esempio del 1987 Tony Hoare suppone di rappresentare in logica del primo ordine il fatto di avere una caramella con il predicato *candy* e con *$1* il possesso di un dollaro, per esprimere l'atto di compravendita di una caramella possiamo scrivere $\$1 \rightarrow candy$.

$$\frac{\$1 \rightarrow candy \quad \$1}{candy} \text{ modus ponens}$$

Se ne deduce $\$1 \wedge candy$ e si ha una caramella senza aver pagato, si può evitare questo problema con modellazioni differenti della base di conoscenza ma si incorre nel *frame problem* [1].

Nella tabella 2.1 è presentata una corrispondenza tra alcuni sistemi substrutturali e le regole in esso valide: in un sistema di tipi rilevante dal momento che non vale la regola di weakening non è possibile avere risorse superflue e ognuna dovrà essere

[1] In intelligenza artificiale è il problema di dover rappresentare una conoscenza in logica senza ricorrere a numerosi assiomi i quali indicano solo che l'ambiente non cambia arbitrariamente.

Tabella 2.1. Sistemi di tipi substrutturali.

	Exchange	Weakening	Contraction
Rilevante	✓	✗	✓
Affine	✓	✓	✗
Lineare	✓	✗	✗

usata almeno una volta. La definizione del verbo "usare" è ambigua e dipende dall'implementazione del linguaggio, si può definire un insieme di azioni e quando si afferma di usare una risorsa ci si riferisce all'esecuzione su questa di una delle azioni possibili, in Rust ad esempio sono specificate dalle regole di ownership.

In un sistema affine non è valida la contraction, questo implica che due usi differenti della stessa risorsa non possano essere unificati in uno solo e in un programma si può tradurre con l'impossibilità di liberare due volte la stessa zona di memoria (4.3); un sistema o una logica lineare [4] è sintesi di uno rilevante e un affine quindi vale solo exchange, ogni risorsa deve essere usata esattamente una volta. Nella sintassi di una logica lineare sono presenti dei qualificatori che marcano una variabile come lineare indicando che debba essere usata una volta.

In un sistema lineare per garantire la non riusabilità di una variabile vengono imposte due invarianti [20]:

1. Le variabili lineari sono usate esattamente una volta per ogni cammino nel diagramma di flusso.

2. Espressioni meno restrittive non possono contenere espressioni più restrittive.

Il perché la prima invariante sia fondamentale è chiaro, dopotutto è proprio il nostro obiettivo; per la seconda supponiamo di avere un oggetto `X` non lineare e un suo attributo `X.y` lineare, sarebbe possibile sfruttare `X` per usare più volte `X.y`, si avrebbe in questo modo una violazione della linearità.

2.2 Linearità in Rust

Rust presenta un sistema dei tipi lineare e tutte le variabili sono marcate come lineari tranne quelle di tipo primitivo, le invarianti sulla linearità vengono rispettate tramite ownership e borrowing.

L'ownership rappresenta il possesso di un right value da parte di un left value (definizione 2.1), in Rust gli oggetti di tipo composto sono lineari e solo a questi si applica ownership, al momento dell'inizializzazione di un oggetto la sua variabile, se presente, è l'unico owner e un assegnamento ad una seconda variabile fa passare il possesso dalla prima alla seconda: in qualsiasi momento quindi può esserci solo un possessore per ogni oggetto. Un owner viene invalidato con la sua distruzione quando esce dal suo scope, bisogna porre particolare attenzione in questo caso dato che le chiamate di funzioni possono invalidare una variabile passata come argomento: il possesso passa alla nuova variabile nel corpo della funzione e una volta terminato il suo scope non sarà più possibile accedere all'oggetto con la variabile vecchia, in quanto si può avere massimo un owner per volta, né con la nuova poiché ormai si è fuori dal suo scope (listato 2.1).

```rust
fn destroy(c: Box<i32>) {
    println!("c: {}", c); // "c: 13"
} // c esce dal suo scope

fn main() {
    let a = Box::new(13i32);
    println!("a: {}", a); // "a: 13"

    let b = a;
    //println!("a: {}", a); // ERRORE! a non e' piu' valida
    println!("b: {}", b); // "b: 13"

    destroy(b);

    //println!("b: {}", b); // ERRORE! b non e' piu' valida
}
```

Listing 2.1. Ownership

Left e Right value

Definizione 2.1. Dato un assegnamento un left value punta ad un'area di memoria contenente un right value, il nome è dovuto alla posizione dei due valori rispetto all'operatore di assegnamento: in `x = y` il left value è `x` e il right `y`.

L'ownership è fondamentale per garantire una gestione automatica e sicura della memoria senza garbage collector ma non è sempre desiderabile prendere possesso di un oggetto, in questi casi si ricorre al borrowing.

Tramite il borrowing si possono creare molteplici riferimenti [9, 4.2] ad un dato purché in sola lettura, con il modificatore `mut` è possibile rendere il riferimento modificabile a patto che in qualsiasi momento per qualsiasi variabile ve ne sia al più uno. Un modulo chiamato **borrow checker** garantisce a tempo di compilazione che finché esistono dei riferimenti ad un oggetto non sia possibile distruggere l'oggetto in questione.

```rust
fn borrow(c: &Box<i32>) {
    println!("c: {}", c); // "c: 13"
} // c esce dal suo scope

fn main() {
    let a = Box::new(13i32);
    let b = &a;

    println!("b: {}", b); // "b: 13"
    borrow(b); // "c: 13"
    println!("a: {}", a); // "a: 13"

}
```

Listing 2.2. Borrowing

Un lifetime è un costrutto usato dal borrow checker per garantire che ogni riferimento sia valido, il lifetime di una variabile inizia quando questa viene creata e

termina quando viene distrutta; il compilatore quando possibile inferisce i lifetime, in caso contrario è necessario gestirli manualmente.

I lifetime sono delle regioni di codice nella quale ogni riferimento deve essere valido [15, 3.3] ma è facile confondere scope e lifetime, negli esempi più semplici coincidono ma i secondi esprimono relazioni tra scope: un lifetime può essere associato a più scope e uno scope può avere associati più lifetime.

Per quanto concerne i lifetime in Rust valgono tre regole assiomatiche [10]:

2.2.1 (Association) *Dato un riferimento il suo scope è sottoinsieme del suo lifetime.*

$$\texttt{x:\&'a T} \longrightarrow scope(x) \subseteq {}'a$$

2.2.2 (Reference) *Un lifetime associato ad un riferimento è sottoinsieme dello scope dell'oggetto cui fa riferimento.*

$$\texttt{x:\&'a T = \&y} \longrightarrow {}'a \subseteq scope(y)$$

2.2.3 (Assignment) *Un lifetime associato ad un riferimento è sottoinsieme del lifetime dell'oggetto cui fa riferimento.*

$$\texttt{x:\&'a S = y: \&'b T} \longrightarrow {}'a \subseteq {}'b$$

e ne deriva una quarta regola:

2.2.4 (Struct reference) *Data una struct* `struct S<'a>{x:&'a T}` *il lifetime associato ad un riferimento di una struct è sottoinsieme del lifetime del membro della struct*[2].

$$\texttt{s:\&'b S<'a>} \longrightarrow {}'b \subseteq {}'a$$

Dimostrazione: Per `s:&'b S<'a>` esiste un `y:S<'a>` tale che `s:&'b S = &y` e applicando 2.2.2 e 2.2.1 si ha $'b \subseteq scope(y) = scope(y.x) \subseteq {}'a$.

e si hanno delle regole speciali:

- Il lifetime speciale `'static` rappresenta lo scope globale, ogni lifetime è suo sottoinsieme $'a \subseteq {}'static$
- La **coercion** permette di forzare un lifetime più lungo in uno più breve `'a:'b` $\leftrightarrow' a \subseteq {}'b$

Si parla di **outliving** quando per una variabile è possibile mantenere il riferimento di un oggetto distrutto. Nelle righe 2 e 5 del listato 2.3 si possono applicare rispettivamente le regole 2.2.1 e 2.2.2 per dedurre $scope(x) \subseteq {}'a$ e $'a \subseteq scope(y)$, è noto $scope(y) \subset scope(x)$ quindi il borrow checker interviene.

```
1 fn main() {
2     let x;
3     {
4         let y = 13;
5         x = &y;
6     }
7     println!("{}", x);
8 }
```

[2] Nel caso in cui dovessero esserci più elementi con lifetime differenti si può considerare il maggiore.

```
error[E0597]: ‘y‘ does not live long enough
 --> life_1.rs:5:13
  |
5 |         x = &y;
  |             ^^ borrowed value does not live long enough
6 |     }
  |     - ‘y‘ dropped here while still borrowed
7 |     println!("{}", x);
  |                    - borrow later used here

error: aborting due to previous error

For more information about this error, try ‘rustc --explain E0597‘.
```

Listing 2.3. Outliving e borrow checker.

Si può ottenere outliving cambiando il tipo di `y` da `i32` a `&i32` (listato 2.4), così facendo non è possibile applicare 2.2.2 e vale $scope(y) \subset scope(x) \subset {}'a$. Un comportamento del genere è inaspettato ma è stato reso possibile per semplificare la definizione di variabili costanti statiche di tipo primitivo [5], i riferimenti ai letterali vengono allocati in memoria di default staticamente e non automaticamente (capitolo 4) creando un riferimento con lifetime `'static`, dunque ogni riferimento ad un dato di tipo primitivo è valido per la durata di tutto il programma.

```
1 fn main() {
2     let x;
3     {
4         let y = &13;
5         x = y;
6     }
7     println!("{}", x);
8 }
```

```
13
```

Listing 2.4. Outliving.

In Rust i tipi composti sono costituiti da una struct, nel listato 2.5 si può vedere la situazione precedente ma con un tipo non primitivo: nella riga 2 si ha $scope(x) \subseteq {}'a$ per 2.2.1 e ${}'a \subseteq scope(y) \subseteq {}'b$ per 2.2.4, è noto $scope(y) \subset scope(x)$ ed essendoci una contraddizione il borrow checker interviene.

```
1 fn main() {
2     let x: &Box<i32>;
3     {
4         let y = &Box::new(13);
5         x = y;
6     }
7     println!("{}", x);
8 }
```

```
error[E0716]: temporary value dropped while borrowed
 --> life_1_2.rs:4:18
  |
4 |         let y = &Box::new(13);
  |                  ^^^^^^^^^^^^ creates a temporary which is freed
    while still in use
```

```
5 |          x = y;
6 |      }
  |      - temporary value is freed at the end of this statement
7 |      println!("{}", x);
  |                     - borrow later used here
  |
  = note: consider using a `let` binding to create a longer lived
    value

error: aborting due to previous error

For more information about this error, try `rustc --explain E0716`.
```

Listing 2.5. Outliving con tipi composti.

Nell'errore in output si parla di **temporary**: è una variabile anonima inizializzata con il risultato di un'espressione, verrà usata per le successive valutazioni e perderà validità solo al termine del suo scope come qualsiasi variabile. A `Box::new(13)` corrisponde una temporary e quando si tenta di leggerne il valore fuori dal suo scope il compilatore segnala l'errore.

Negli esempi precedenti i lifetime sono sempre stati inferiti ma nel listato 2.6 sono esplicitati, si ha:

$$scope(rs1) \subseteq {}'a \subseteq scope(s1)$$
$$scope(rs2) \subseteq {}'a \subseteq scope(s2)$$
$$scope(result) \subseteq {}'a$$

che è soddisfatto con $'a = scope(result)$.

```
1  fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2      if x.len() > y.len() { x }
3      else { y }
4  }
5
6  fn main() {
7      let s1 = String::from("short");
8      let s2 = String::from("long");
9      let result;
10     {
11         let rs1 = &s1;
12         let rs2 = &s2;
13         result = longest(rs1, rs2);
14     }
15     println!("The longest string is {}", result);
16 }
```

```
The longest string is short
```

Listing 2.6. Lifetime espliciti.

Quando a seguito di un assegnamento la variabile memorizza un riferimento ad un oggetto si parla di **shallow copy**, se invece si memorizza il dato per intero si ha una una **deep copy**. In Rust per i dati di tipo primitivo si eseguono sempre deep copy mentre per gli oggetti composti si eseguono di default shallow copy, è

comunque possibile copiare l'oggetto nella sua interezza tramite il **trait** (simile a quello che in Java è un'interfaccia) `Clone`; questo avviene poiché di un dato di tipo primitivo la dimensione è nota a tempo di compilazione ma non si può dire lo stesso degli oggetti composti. Nel listato 2.7 `x` e `y` puntano a due locazioni differenti.

```
fn main () {
    let x = 13;
    let x = y; // deep copy
}
```

Listing 2.7. Ownership con tipi primitivi.

Se con ownership e borrowing l'esecuzione di un programma è molto più rapida e sicura il linguaggio è più verboso e richiede maggior impegno cognitivo da parte del programmatore.

Capitolo 3

Security policy

Risulta molto difficile parlare di sicurezza in generale dato che non ne esiste una definizione assoluta ma 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** ovvero un documento contenente regole, principi e pratiche che determinano come garantire che il sistema si trovi in uno stato sicuro [11].

Security policy

Definizione 3.1. In una politica di sicurezza si definiscono quali azioni i **principal** possono eseguire sugli oggetti. Un principal è un'entità qualsiasi che interagisce con il sistema e deve rispettare la politica di sicurezza, nel caso specifico dei linguaggi di programmazione saranno entità software.

Nello sviluppo di linguaggi di programmazione sicuri si è guidati da due principi [12]:

- **Trusted computing base (TCB)**. Un sistema presenta componenti critici per il rispetto della policy, questi formano il TCB ed è fondamentale non presenti vulnerabilità dato che metterebbero a repentaglio la sicurezza del sistema stesso, il TCB deve essere semplice e limitato così da poterne verificare facilmente la correttezza ed evitare fallimenti.
- **Principle of Least Privilege (POLP)**. Durante l'esecuzione ogni entità deve avere il numero minore possibile di permessi necessari per eseguire il suo compito.

È possibile scomporre i requisiti di sicurezza fondamentali durante lo sviluppo in **access control** e **information flow**, il primo limita chi o cosa possa accedere a quali risorse e il secondo definisce quali operazioni siano corrette o meno in seguito ad un accesso conforme all'accesso control, di norma il sistema operativo stesso implementa questi controlli via software o hardware e programmi che sono legali in C potrebbero comunque essere interrotti. In Rust la sicurezza è garantita da regole semplici e verificabili e si rispetta il POLP, i controlli sono eseguiti a tempo di compilazione e anche quando un programma viene interrotto a runtime non viene scomodato il sistema operativo (vedere ad esempio il codice nel listato 4.4).

Questa necessità di implementare la sicurezza nel linguaggio stesso e non rimetterla più nelle mani del sistema operativo è nata dall'avvento di internet e la sempre più grande condivisione di codice potenzialmente pericoloso.

Per dimostrare delle proprietà di un linguaggio serve una definizione formale della sua **semantica** ovvero l'insieme delle regole che definiscono il significato di un linguaggio e dei suoi programmi. A tal proposito Rust non ha una semantica formale definita (figura 3.1) e la sua definizione di sicurezza è poco chiara, un programma è sicuro se rispetta **memory safety** (capitolo 4) e **semantica statica** (capitolo 5) [18, 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." [18, 14.3]

"Del codice in Rust non è corretto se presenta almeno uno dei comportamenti in lista. Questo vale anche per codice in blocchi `unsafe` e funzioni `unsafe`. `unsafe` significa solo che evitare undefined behaviour è compito del programmatore; non cambia alcuna garanzia sul fatto che i programmi in Rust non debbano mai causare undefined behaviour."

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 "sottolinguaggi": *unsafe Rust* e *safe Rust* in base alla presenza o meno di un blocco `unsafe` nel codice, questo costrutto è desiderabile perché permette sia di poter comunicare con librerie in *C* che di avere un accesso più esplicito alla memoria [15, 1.2]; piuttosto che una definizione formale di undefined behavior si ha una lista in continuo aggiornamento [18, 14.3], è importante notare che comunque safe Rust presenta del codice in blocchi `unsafe` scritto dagli sviluppatori.

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 safe Rust, 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 3.1 si ha un **raw pointer dereferencing** (presente in lista come dimostrato in figura 3.1): è possibile leggere il valore memorizzato in una locazione di memoria tramite il suo indirizzo come in C.

È ragionevole che questo sia possibile solo in unsafe Rust ma mette in luce come la documentazione sia poco chiara.

```
1 fn main() {
2     unsafe {
3         let x = 42;
4         let ptr = &x as *const i32;
5         println!("Dereferencing: {}", *ptr);
6     }
```

```
}

Dereferencing: 42
```

Listing 3.1. Raw pointer dereferencing in un blocco `unsafe`

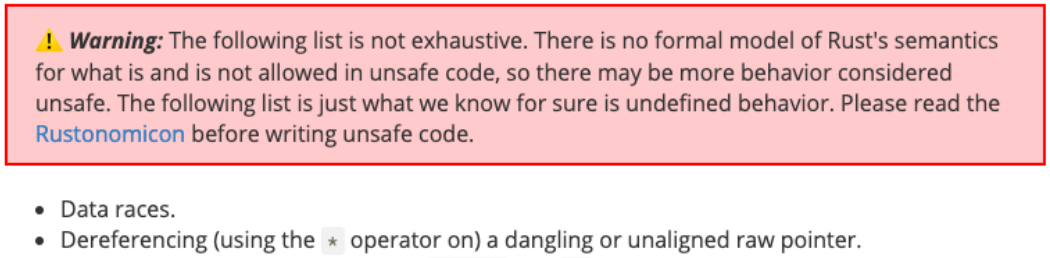

Figura 3.1. Il raw pointer dereferencing è considerato undefined behavior

Capitolo 4

Memory Safety

Durante l'esecuzione di un programma ogni oggetto ha associato un insieme finito di locazioni di memoria cui accedere noto come **address space**, se in un programma nessun oggetto accede mai ad indirizzi fuori dal proprio address space allora è **memory safe** e un linguaggio che garantisce la memory safety dei suoi programmi è detto a sua volta memory safe.

L'address space può essere definito a tempo di compilazione o esecuzione dipendentemente dal tipo di allocazione, in C e Rust ne consideriamo tre tipi:

- **Allocazione statica**. Il compilatore memorizza solo ed esclusivamente le variabili globali ovvero quelle il cui scope corrisponde a tutto il file, si possono definire tramite un modificatore (in C è `static`, da cui il nome) o definendole fuori da ogni funzione.
- **Allocazione automatica**. Si usa per variabili locali e di tipo primitivo. Una variabile è locale quando dichiarata all'interno di una funzione (quindi valida solo in un determinato scope), dal momento che l'ordine di esecuzione non è noto a tempo di compilazione l'allocazione avviene a tempo di esecuzione e si utilizza una struttura dati nota come **stack** o **call stack**. Spesso il nome call stack viene utilizzato per le sole chiamate di funzione.
- **Allocazione dinamica**. L'allocazione dinamica in C è esplicita tramite funzioni come `malloc` e permette di allocare memoria ad un oggetto durante l'esecuzione, si rivela particolarmente utile con oggetti che non hanno dimensione fissa. In safe Rust non si hanno funzioni del genere per questioni di sicurezza ma si ha il tipo `Box`, si usa una struttura dati chiamata **heap** che ha la particolarità di crescere verso l'alto, contrariamente allo stack che cresce verso il basso, e può essere problematico in caso di overflow (sezione 4.1). Un tipo composto è definito dall'unione di tipi primitivi e non è banale prevedere la dimensione di un oggetto composto quindi vengono allocati sull'heap. Un oggetto di tipo primitivo viene salvato sullo stack.

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 è presente in Rust.

4.1 Buffer Overflow

Un **buffer** è una qualsiasi zona contigua di memoria contenente istanze dello stesso dato e ne definisce i limiti, se durante l'esecuzione si riescono a superare si parla di **buffer overflow** ed è possibile leggere o scrivere oltre il proprio address space. Un esempio di buffer overflow in C è nel listato 4.1 [16, 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 4.2 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);
}
```

```
warning: this program uses gets(), which is unsafe.
START
str1:START, str2:START, valid:1
```

```
warning: this program uses gets(), which is unsafe.
EVILINPUTVALUE
str1:TVALUE, str2:EVILINPUTVALUE, valid:0
```

```
warning: this program uses gets(), which is unsafe.
BADINPUTBADINPUT
str1:BADINPUT, str2:BADINPUTBADINPUT, valid:
```

Listing 4.1. Buffer overflow in C

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

```
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 4.2. Buffer overflow in Rust

La pericolosità del buffer overflow è dovuta alla possibilità di attuare una **code injection** ovvero eseguire codice arbitrario, nel listato 4.3 se ne ha un esempio molto semplice, per la stringa in input non si ha alcun tipo di bound check ed è possibile scrivere un comando che verrà eseguito da `system()` con gli stessi permessi con cui si esegue il file.

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

int main(int argc, char **argv) {
 char cat[] = "cat ";
 char *command;
 size_t commandLength;

 commandLength = strlen(cat) + strlen(argv[1]) + 1;
 command = (char *) malloc(commandLength);
 strncpy(command, cat, commandLength);
 strncat(command, argv[1], (commandLength - strlen(cat)) );

 system(command);
 return (0);
}
```

```
Nel mezzo del cammin di nostra vita...
$ ./bin/inj "file.txt; ls"
Nel mezzo del cammin di nostra vita...
bin             file.txt        inj.c           output
```

Listing 4.3. Code injection in C

Esistono code injection più sofisticate che possono eseguire codice arbitrario sfruttando le istruzioni in assembly di un programma e la rappresentazione in memoria di un processo [1], questo è costituito da tre regioni:

- **Text**. Contiene le istruzioni da eseguire e dati in sola lettura, quest'area è read-only quindi un tentativo di scrittura è intercettato dal sistema operativo.
- **Data**. Contiene dati con scope globale, è l'area di memoria usata da un'allocazione statica.
- **Stack**. Lo stack sul quale si salvano i dati con allocazione automatica. Proprio grazie allo stack si può sfruttare un buffer overflow per puntare ad un processo differente ed eseguire codice non desiderato.

4.2 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 e ognuno è 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**. Si applica un clamp ovvero dati min e max - rispettivamente il numero più grande e più piccolo rappresentabile - e un numero n in memoria si salva

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

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 in quanto 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 i quali controlli dinamici per l'integer overflow (listato 4.4) 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);
    }
}
```

```
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 4.4. Integer overflow in Rust

Nel listato 4.5 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 (uguale a 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>
```

```
4
5  int main(int argc, char *argv[]){
6      unsigned short s;
7      int i;
8      char buf[80];
9
10     if(argc < 3) return -1;
11     i = atoi(argv[1]);
12     s = i;
13
14     if(s >= 80) return -1;
15
16     printf("s = %d\n", s);
17
18     memcpy(buf, argv[2], i);
19     buf[i] = '\0';
20     printf("%s\n", buf);
21
22     return 0;
23 }
```

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

Listing 4.5. Integer overflow in C

4.3 Double-Free

Si ha un double free error [6, 10.4.4] quando si prova a liberare più volte la stessa zona di memoria e in Rust non è possibile grazie all'ownership. In C invece si può osservare il codice nel listato 4.6: 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, il risultato è che Luisa non ha più un account e a quello di Carla possono accedere sia Guido che Luisa.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct User User;
6
7  struct User {
8      User * self;
9      char name[8];
10 };
11
12 int main(int argc, char** argv) {
13
14     // (1) Guido si iscrive
15     struct User * Guido = malloc(sizeof(struct User));
16     Guido->self = Guido;
17     strcpy(Guido->name, "Guido");
18
```

```
    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
    );
}
```

```
Guido: [self: 0x7f9f72c05a30, name: Guido]
Luisa: [self: 0x7f9f72c05a30, name: Luisa]
Carla: [self: 0x7f9f72c05a30, name: Carla]
Luisa: [self: 0x7f9f72c05a30, name: Carla]
```

Listing 4.6. Double free in C

Il puntatore `Guido` è anche un **dangling pointer** (sezione 4.4).

4.4 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 si 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

4.7 l'indirizzo viene stampato a schermo e inserito dall'utente, indicato da `>`, per semplicità.

```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 * 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);
}

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);
}
```

```
0x7fec9fe04d10
> 0x7fec9fe04d10
[Guido, Carla, asa nisi masa]
```

Listing 4.7. Dangling pointer in C

In Rust non è possibile ricreare l'esempio in C perché come già visto il borrow checker intercetta outliving con struct, è però possibile avere outliving con tipi primitivi come già visto, nel listato 4.8 c'è un esempio con stringhe.

```rust
fn main() {
    let s1;
    {
        let s2 = &"asa nisi masa";
        s1 = s2;
    }
    println!{"{}", s1};
}
```

```
asa nisi masa
```

Listing 4.8. Dangling pointer in Rust

Capitolo 5

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. In informatica un tipo è un vincolo che definisce un insieme ben definito di valori che la risorsa di un programma può assumere. Gli errori riscontrabili in fase di esecuzione possono essere **trapped** o **untrapped**, i primi contrariamente ai secondi sono facilmente riconoscibili dato che interrompono l'esecuzione del programma, gli untrapped si rivelano più insidiosi dal momento che potrebbero eseguire codice non voluto. Un programma che supera il controllo dei tipi (definizione 5.1) è ben tipato e un linguaggio in cui nessun programma ben tipato genera errori durante l'esecuzione si dice **sound**. In un linguaggio **safe** nessun programma genera errori untrapped e un linguaggio sound è necessariamente type safe [3]. Una definizione analoga ma meno formale di Milner spesso adottata è la seguente:

well-typed programs cannot "go wrong" [11]

con programmi che "vanno male" si possono intendere programmi non sound o che generano undefined behavior (capitolo 3).

Type checking

Definizione 5.1. È la verifica dei vincoli imposti dal sistema dei tipi di un linguaggio. Il type checking può avvenire sia a tempo di compilazione che a tempo di esecuzione e si parla rispettivamente di type checking statico/forte e dinamico/debole. Un controllo statico è desiderabile per l'ottimizzazione che ne consegue ma non sempre è possibile verificare la correttezza di un programma a tempo di compilazione, di norma sono utilizzati sia static che dynamic type checker, in Java ad esempio i controlli sui metodi sono eseguiti dinamicamente.
Un programma che supera la fase di type checking è ben tipato.

In un linguaggio tipato l'insieme delle regole che assegna un tipo ad ogni espressione è detto semantica statica o **sistema dei tipi**, in [8] vengono identificate 4 componenti che permettono di definire un linguaggio più sicuro, queste sono **type safety**, **type casting**, **type initialization** e **immutability**.

5.1 Type safety

5.2 Type casting

Quando si esegue un'operazione tra oggetti di tipo differente può rivelarsi necessaria una conversione di tipo (type casting) che può essere esplicita o implicita, nel secondo caso si parla di **coercion**. Tipicamente i linguaggi con type checking forte eseguono pochi casting impliciti e quelli con type checking dinamico molti. Rust è fortemente tipato e non c'è coercion, C è debolmente tipato e presenta conversione implicita.

```c
#include <stdio.h>

int main() {
    int x = 12;
    printf("%f\n", x * 0.5);
}
```

```
6.000000
```

Listing 5.1. Casting implicito in C

```rust
fn main() {
    let x = 12;
    // println!("{}", x * 0.5); // ERRORE!
    println!("{}", x as f32 * 0.5);
}
```

```
6
```

Listing 5.2. Casting in Rust

Rust non ha casting implicito perché può portare a vulnerabilità, un esempio di questa vulnerabilità è un bug del 2002 nel sistema operativo FreeBSD per cui in alcune chiamate di sistema si assumeva a priori che il valore in input fosse un intero positivo nonostante nella definizione della funzione il tipo fosse `int`. I programmatori data l'assunzione fatta non eseguivano boundary check, questo valore veniva poi passato a `memcpy()` che prende un `unsigned int` e a causa della coercion un valore come -1 veniva convertito in $2^{32} - 1$, valore che non avrebbe superato il boundary check. Inserendo numeri negativi abbastanza grandi era possibile copiare una porzione di memoria riservata al kernel [17]. Per chiarezza nel listato 5.2 è stato riproposto del codice a grandi linee, il problema è la conversione di `maxlen` che supera il controllo a riga 11 e viene convertita da intero a intero positivo a riga 12.

```c
#include <stdlib.h>

#define KSIZE 1024

char kbuf[KSIZE];

void *memcpy(void *dest, void *src, size_t n);

int copy_from_kernel(void *user_dest, int maxlen) {
    // len=min(KSIZE, maxlen)
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
```

```
    return len;
}
```

5.3 Type initialization

La dichiarazione di una variabile in C le assegna un address space ma non c'è inizializzazione, il valore della variabile sarà spesso privo di significato o addirttura illegibile poiché corrisponde al dato inserito in quella stessa locazione da un processo passato e riconvertito nel tipo della nuova variabile.

Ad oggi un comportamento del genere è un caso raro e di norma ad una variabile appena dichiarata viene assegnato un valore di default, ad esempio 0 per i numeri, la stringa vuota per le stringhe e `NULL` per gli oggetti.

Il valore `NULL` fu introdotto da Tony Hoare nel 1964 e nel 2009 lui stesso lo definì il suo errore da un miliardo di dollari [7] perché ha portato a numerose sviste ed errori da parte dei programmatori: in C compilando con *gcc* un puntatore a `NULL` punta a una locazione 0 riservata (listato 5.3) e la sua dereferenza causa l'interruzione del programma.

```c
#include <stdio.h>

int main() {
    void* ptr = NULL;
    printf("%p\n", ptr);
    printf("%d\n", * (int *) ptr);
}
```

```
0x0
[1]    89833 segmentation fault  ./bin/null_c
```

Listing 5.3. Null reference in C

In Rust non esiste `NULL` ma si ha il tipo polimorfo `Option<T>` che può assumere i valori `Some<T>` o `None`. Nel listato 5.4 si può vedere come un `Box<i32>` senza alcun valore punti ad un `None` di tipo `Option<Box<i32>>` e non ad un valore speciale `NULL`, il quale si colloca meno coerentemente nel sistema dei tipi.

```rust
fn check_optional(optional: Option<Box<i32>>) {
    match optional {
        Some(p) => println!("Si ha il valore {}", p),
        None => println!("Non si ha alcun valore"),
    }
}

fn main() {
    let optional = None;
    check_optional(optional);
    let optional = Some(Box::new(42));
    check_optional(optional);
}
```

```
Non si ha alcun valore
Si ha il valore 42
```

Listing 5.4. Null reference in Rust

5.4 Immutability

Un oggetto è immutabile quando durante l'esecuzione di un programma non può essere modificato, in Rust ogni oggetto è di default immutabile ma si possono definire esplicitamente oggetti il cui valore è modificabile tramite la parola chiave `mut`, questa scelta è coerente con il principio di least privilege. Le costanti invece sono definite tramite la parola chiave `const` al posto del `let`, una costante deve essere nota a tempo di compilazione quindi è necessario indicarne esplicitamente il tipo e il valore, quest'ultimo deve essere un'espressione costante e non il risultato di una valutazione di qualche espressione eseguita a runtime. È possibile definire una costante in qualsiasi parte del codice ma il lifetime sarà sempre `'static`, non è possibile definire una costante `mut` [9, 3.1].

In C non c'è una vera e propria immutabilità, tramite il modificatore `const` è possibile definire delle costanti che dovrebbero essere non modificabili, rimane comunque possibile dereferenziarle e tramite un puntatore modificarne il valore, l'immodificabilità assume particolare rilievo in programmi concorrenti. Nel listato 5.5 abbiamo:

1. `Misc`. Una struct composta da un intero, il cui valore non è rilevante, e un puntatore a un intero nel quale verrà salvato l'indirizzo di `accum`.

2. `accum`. Una variabile globale costante, il segmento in cui verrà allocata dipende dal compilatore ad esempio usando *gcc* è salvato nel text segment del processo insieme al codice ma in ogni caso è un'area in sola lettura.

3. `accum_mutex` .Un **mutex** o **lock** è un meccanismo di sincronizzazione per imporre la mutua esclusione ed evitare **race condition**, garantisce quindi che la stessa risorsa non venga modificata da più thread contemporaneamente. Il possesso e il rilascio sono effettuati mediante le due funzioni di lock e unlock.

4. `ths`. Un array di 20 thread ovvero sottoprocessi eseguiti parallelamente possibilmente condividendo delle risorse. Tramite la chiamata a `pthread_join()` si attende il termine del thread in input memorizzandone il risultato di ritorno in `ret`.

L'esecuzione del programma è interrotta dal sistema operativo perché `accum` è read-only ma in C non c'è alcun controllo.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  typedef struct {
6      int v;
7      int* ptr;
8  }Misc;
9
10 // global
11 const int accum = 0;
12 pthread_mutex_t accum_mutex = PTHREAD_MUTEX_INITIALIZER;
13
14 // func
```

```
15 void* func(void* x) {
16     Misc* xm = (Misc*) x;
17
18     pthread_mutex_lock(&accum_mutex);
19     *xm->ptr += xm->v;
20     pthread_mutex_unlock(&accum_mutex);
21     return NULL;
22 }
23
24 int main() {
25     pthread_t ths[20];
26     Misc misc[20];
27     int i;
28
29     for(i=0; i < 20; i++) {
30         misc[i].v = i;
31         misc[i].ptr = &accum;
32
33         pthread_create(&ths[i], NULL, func, &misc[i]);
34     }
35
36     for(i=0; i < 20; i++) {
37         void* res;
38         pthread_join(ths[i], &res);
39     }
40     printf("accum = %d\n", accum);
41 }
```

```
thread.c:19:10: warning: initializing 'int *' with an expression of
    type 'const int *' discards qualifiers [-Wincompatible-pointer-
    types-discards-qualifiers]
     int* ptr = &accum;
          ^     ~~~~~~
1 warning generated.
$ ./bin/thread_c
[1]    76864 bus error  ./bin/thread_c
```

Listing 5.5. Costanti e thread in C

Il programma nel listato 5.6 funziona perfettamente, in safe Rust non c'è il pericolo che un valore immutabile venga modificato da un riferimento perché non è possibile dereferenziare un puntatore, vale la pena però notare che in tutto il programma non compaia mai il modificatore `mut`. La funzione `thread::spawn()` prende in input una **closure**, è come una funzione anonima in Java e all'interno delle doppie pipe `||` si inseriscono i parametri in input, in questo caso nessuno. La parola chiave `move` dà alla closure l'ownership delle variabili che usa evitando così che un thread (`main` compreso) modifichi una risorsa usata da un altro oggetto, solo una variabile per volta potrà avere il possesso della risorsa in questione, si garantisce la mutua esclusione e si evitano race condition a tempo di compilazione [14, 5.6]. Per la sincronizzazione tra thread si utilizzano le funzioni `channel`.

```
1 use std::thread;
2 use std::sync::mpsc;
3
4 fn main() {
5     // tx is the transmitter or sender
```

```
    // rx is the receiver
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = i * i;

            tx.send(answer).unwrap();
        });
    }

    for _ in 0..10 {
        println!("{}", rx.recv().unwrap());
    }
}
```

```
0
1
4
9
16
25
36
49
81
64
```

Listing 5.6. Concorrenza in Rust

5.5 Rapporto con Memory Safety

Type safety e memory safety sono strettamente collegate tanto che a volte risulta davvero difficile distinguere tra le due e alcune vulnerabilità violano entrambe, ad esempio nel listato 5.7 si alloca un array di `short` (2 B) sullo stack, considerando che il computer su cui ho eseguito il programma monta un processore *Intel* (little endian quindi la cifra più a destra è la meno significativa) 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 $2^{16} = 65536$. La causa di questa confusione sia la lettura di un solo oggetto da 4 B anziché due oggetti da 2, questo avviene a causa della rappresentazione di due tipi differenti e del casting che C permette contrariamente a Rust.

```c
#include <stdio.h>

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

```
65536
```

Listing 5.7. Type confusion in C

Capitolo 6

Conclusioni

Bibliografia

[1] Aleph One. Smashing the stack for fun and profit. *http://www. shmoo. com/phrack/Phrack49/p49-14*, (1996).

[2] Barr, M. Real men program in C (2009). Available from: `https://www.embedded.com/real-men-program-in-c/`.

[3] Cenciarelli, P. Dispense del corso linguaggi di programmazione (2019). Available from: `http://wwwusers.di.uniroma1.it/~lpara/DISPENSE/note.pdf`.

[4] Girard, J.-Y. Linear logic. *Theoretical Computer Science*, **50** (1987), 1 . Available from: `http://www.sciencedirect.com/science/article/pii/0304397587900454`, `doi:https://doi.org/10.1016/0304-3975(87)90045-4`.

[5] rust-lang/rfcs. Available from: `https://github.com/rust-lang/rfcs/blob/master/text/1414-rvalue_static_promotion.md`.

[6] Gollmann, D. *Computer Security*. John Wiley & Sons, Inc., USA (1999). ISBN 0471978442.

[7] Hoare, T. Null references: The billion dollar mistake (2009). Available from: `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`.

[8] Khwaja, A., Murtaza, M., and Ahmad, H. A security feature framework for programming languages to minimize application layer vulnerabilities. *Security and Privacy*, **3** (2019). Available from: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spy2.95`, `doi:10.1002/spy2.95`.

[9] Klabnik, S. and Nichols, C. The Rust programming language (2020). Available from: `https://doc.rust-lang.org/book/title-page.html`.

[10] Mechpen. Mis-understanding Rust lifetime (2019). Available from: `https://mechpen.github.io/posts/2019-09-28-rust-lifetime/index.html#example-3`.

[11] Milner, R. A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17** (1978), 348.

[12] Niño, J. Implementing security via modern programming languages (2009). Available from: `http://www.cs.uno.edu/~jaime/plSecurity.pdf`.

[13] Pypl popularity of programming language (2020). Available from: `http://pypl.github.io/PYPL.html`.

[14] The Rust programming language. Available from: `https://doc.rust-lang.org/1.9.0/book/README.html`.

[15] Rustonomicon (2020). Available from: `https://doc.rust-lang.org/nomicon/index.html`.

[16] Stallings, W. *Operating systems: internals and design principles*. Boston: Prentice Hall„ 7 edn. (2012). ISBN 978-0-13-230998.

[17] The FreeBSD Project. Freebsd-sa-02:38.signed-error (2002). Available from: `https://www.freebsd.org/security/advisories/FreeBSD-SA-02:38.signed-error.asc`.

[18] The Rust reference (2020). Available from: `https://doc.rust-lang.org/reference/introduction.html`.

[19] Tiobe programming community index (2020). Available from: `https://www.tiobe.com/tiobe-index/`.

[20] Walker, D. Substructural type systems. *Advanced topics in types and programming languages*, (2005), 3.