ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI SCIENZE

Corso di Laurea in Ingegneria e Scienze Informatiche

PROGRAMMAZIONE MEMORY-SAFE SENZA GARBAGE COLLECTION: IL CASO DEL LINGUAGGIO RUST

Relazione finale in
Programmazione ad Oggetti

Relatore:
Prof. MIRKO VIROLI

Co-relatore:
Ing. DANILO PIANINI

Tesi di Laurea di:
MANUELE PASINI

ANNO ACCADEMICO 2018–2019
SESSIONE II

PAROLE CHIAVE

Rust

Memory-Safety

Ownership

Lifetimes

Borrowing

Indice

1 Introduzione **9**

2 Il linguaggio di programmazione Rust **11**
2.1 Variabili 11
2.1.1 Slice types 12
2.1.2 Shadowing 12
2.2 Funzioni 13
2.3 Strutture dati 14
2.4 Controllo di flusso 16
2.5 Macro 17
2.6 Pattern matching e destructuring declarations 18

3 Gestione della memoria **21**
3.1 Ownership 21
3.2 Borrowing rule 24
3.3 Lifetime 25
3.4 Conclusioni sulla memory-safety 26

4 Type System di Rust **29**
4.1 Generici 29
4.2 Trait 30
4.2.1 Trait bound 30
4.3 Trait object 31
4.4 Coherence and The orphan rule 32
4.5 Chiusure 33
4.6 Smart pointers 33
4.6.1 Interior Mutability Pattern 34

5 Altri Aspetti **37**
5.1 Struttura di un progetto Rust 37
5.1.1 Testing 39
5.2 Concorrenza in Rust 40

6 Conclusioni **43**
6.1 Rust è un linguaggio di programmazione ad oggetti? 43

7 Ringraziamenti **45**

"And don't you know that God is Pooh Bear? the evening star must be drooping and shedding her sparkler dims on the prairie, which is just before the coming of complete night that blesses the earth, darkens all the rivers, cups the peaks and folds the final shore in, and nobody, nobody knows what's going to happen to anybody besides the forlorn rags of growing old, I think of Dean Moriarty, I even think of Old Dean Moriarty the father we never found, I think of Dean Moriarty."

Capitolo 1

Introduzione

Cosa si intende con memory-safety? Dare una definizione precisa di questo concetto non è semplice; generalmente, con memory-safety si intende l'assenza di un insieme di errori collegati ad un utilizzo improprio della memoria, più specificamente si fa riferimento ad errori di accessi ad aree di memoria non autorizzati, doppie deallocazioni di determinate aree di memoria, accessi ad aree di memoria precedentemente dealloccate e via dicendo. L'assenza di questi errori può portare a definire un programma memory safe?[1] La caratteristica di questi errori, è che hanno conseguenze diverse in base al linguaggio di programmazione nel quale avvengono: all'interno del linguaggio C~[2], la presenza di un errori di questo genere porta il programma ad avere un comportamento indefinito, che spesso varia in base alle scelte del compilatore a run time, creando un ambiente potenzialmente pericoloso. Al contrario in linguaggi come Java~[3], sebbene sia allo stesso modo possibile accedere ad esempio oltre l'indice massimo di un vettore, il comportamento è noto e ben definito: il programma termina. Il problema della memory-safety è un problema importante e ricorrente all'interno di ogni software, basti pensare alle dichiarazioni fatte da Matt Miller, security engeneer di Microsoft, per il quale, da dodici anni a questa parte, il 70% di tutte le vulnerabilità presenti all'interno dei prodotti Microsoft sono dovute a problemi di gestione della memoria. [1] Allo stesso modo, in Gecko, motore di rendering del browser Firefox, circa il 50% degli errori a runtime riguardano l'utilizzo di aree di memoria precedentemente dealloccate, deallocazioni doppie di aree di memoria o errori di accesso ad indici al di fuori dei limiti.~[4]; Gecko è scritto principalmente in C++, linguaggio che per quanto veloce in tempi di esecuzione

[1] `https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf`

è altrettanto non-sicuro e per quanto un programmatore possa essere esperto, errori di questo genere sono inevitabilmente presenti.
Un altro esempio di vulnerabilità nata da problemi di memory-safety è "Heartbleed"~[5], reso pubblico nel 2014, si basa sul un mancato controllo sulla lunghezza delle richieste all'interno del protocollo SSL, mancato controllo che porta chi attacca a poter accedere ad aree di memoria a lui non riservate.

Rust è un linguaggio di programmazione a basso livello nato all'interno di Mozilla nel 2006 come progetto personale di un dipendente dell'azienda, la quale ha abbracciato il progetto nel 2009; la prima versione stabile risale al 2015.[2]

> "Rust is a statically typed systems programming language most heavily inspired by the C and ML families of languages. Like the C family of languages, it gives the developer fine control over memory layout and predictable performance. Unlike C programs, Rust programs are memory safe by default."~[4]

L'obbiettivo di Rust è far si che dal momento in cui la compilazione di un programma avviene con successo e senza alcun errore, allora quest'ultimo non darà problemi durante l'esecuzione: questo goal imposto dagli ideatori di Rust fa si che sia un linguaggio molto rigido, che spesso preferisce rifiutare programmi perfettamente funzionanti in quanto il compilatore non riesce ad averne la certezza; questa rigidità tuttavia fa si che l'avvenuta compilazione sia spesso sinonimo di memory-safety.

[2] `https://blog.rust-lang.org/`

Capitolo 2

Il linguaggio di programmazione Rust

2.1 Variabili

Rust è un linguaggio con typing statico ed imperativo. Generalmente i linguaggi di programmazione hanno istruzioni ed espressioni: viene definita espressione una qualsiasi combinazione di variabili, costanti, valori, operatori e funzioni che produca un risultato; viene definito al contrario istruzione un qualsiasi comando che permetta di svolgere una determinata azione. All'interno di Rust quasi tutto ha un valore e può essere un espressione; un'istruzione termina con il punto e virgola, un'espressione no.~[6]
Ogni variabile all'interno di Rust viene dichiarata tramite la parola chiave `let` che può essere seguita dal tipo di dato: quest'ultimo può essere omesso in quanto viene dedotto dal compilatore tramite type-inference; ogni variabile è immutabile di default, una volta istanziata non è possibile modificarne il valore, in controtendenza rispetto a linguaggi come Java per i quali una variabile è immutabile solo se definita "final". Questa decisione è stata presa presa sulla base del fatto che in progetti di medie/grandi dimensioni è più semplice individuare azioni di scrittura; l'immutabilità delle variabili non compromette l'esistenza di tipi di dato come costanti o variabili statiche. Una variabile può tuttavia essere modificata se, al momento della dichiarazione, viene specificata la parola chiave `mut`.

```
let b : i32 = 15;
let mut a : = 10;
```

2.1.1 Slice types

Rust è dotato di due tipi di stringa: String e &str (string slice). Più in generale uno slice è una vista all'interno di un'area di memoria; viene rappresentato da una tupla puntatore-lunghezza. Lo string slice è l'unico tipo di stringa presente all'interno del core di Rust mentre invece String, che non è altro che un vettore di bytes, è fornita da una libreria.

```
let str_one : String = "I'm string".to_string();
let str_lit : &str  = "I'm a string";
let slice : &str = &str_lit[0..3];
```

Essendo un wrapper del tipo vettore, una stringa può essere espansa tramite push, così come sono funzionanti altri metodi del tipo vettore, ad eccezione dell'indexing, procedura attraverso la quale si accede ad un elemento specifico di una stringa o più in generale di una collezione di elementi tramite la posizione che esso occupa all'interno della collezione stessa. Una stringa non può essere acceduta tramite indice: il tipo String è codificato in UTF-8, codifica che permette l'utilizzo di caratteri speciali come ad esempio il cirillico, tuttavia per rappresentare un tale carattere vengono utilizzati due byte, di conseguenza accedere ad una stringa in indice due ha un comportamento undefined che dipende da che tipo di stringa viene ricevuta a runtime. Non potendo accedere tramite indexing diventa complicato anche estrarre ad esempio una parola dall'interno di una stringa; più in generale gli unici modi con cui scorrere e nel caso estrarre parte di una stringa sono iteratori e slice type. Uno string slice è un riferimento ad una porzione di stringa, può essere ricavato tramite l'uso di parentesi quadre:

```
let a = String::from("my String");
let b = &s[0..4];
```

In questo caso "b" è un riferimento a quella parte di "a" dall'indice 0 all'indice 4. Gli slice type vengono utilizzati soprattutto come parametri di funzioni in modo da non dover prendere l'ownership di una stringa e allo stesso tempo per poterla ricevere come parametro.

2.1.2 Shadowing

Con shadowing si intende la capacità di una variabile di oscurarne un'altra con lo stesso nome identificativo. In linguaggi di programmazione come Java o C++~[7], il concetto di shadowing non è presente; in Rust questo concetto viene introdotto: se in uno scope sono presenti due variabili con lo stesso nome, quella

dichiarata per ultima nasconde la prima.

```
let a = 13;
let a = 27;
println!("{}",a); //Prints 27
```

Qual è dunque la differenza tra dichiarare una variabile mutabile oppure modificarla tramite shadowing? Tramite shadowing, la variabile oscurata cessa di esistere in favore di quella nuova che prende il suo posto. Si rivela utile in condizioni nelle quali non ci interessa tanto il valore di una variabile ma una sua proprietà, ad esempio il numero di caratteri all'interno di una stringa.
Un aspetto importante dello shadowing è la possibilità di cambiare il tipo di una variabile, pur essendo Rust un linguaggio statico.

```
let a = 33;
let a = "I became a string";
```

In questo caso il compilatore assegna inizialmente ad `a` un valore intero, per poi effettuare, la riga dopo, shadowing su essa assegnandole uno slice.

2.2 Funzioni

Le funzioni vengono dichiarate seguendo lo stile di linguaggi come Pascal~[8], dove il tipo segue il nome (sia della funzione, che dei parametri). Ogni funzione deve avere un valore di ritorno, che si tratti di un tipo noto o di uno unit type `()`.

```
fn foo(par1 : i32) -> () {
    //Do nothing
}
```

Lo unit type può essere associato al tipo `void` di linguaggi di programmazione di alto livello come Java, sebbene nel concreto mentre il tipo "unit" assume un singolo valore, per l'appunto `()`, il tipo `void` di linguaggi C-like non assume nessun valore. Viene usato principalmente per gestire l'esito di procedure.

```
fn add_foo(par1:i32 , par2: i32) -> i32{
    par1 + par2
}
```

In questo caso, siccome è necessario avere come valore di ritorno un intero con segno a trentadue bit, è necessaria l'utilizzo di un'espressione. Può succedere che una funzione per qualche motivo particolare non riesca ad eseguire il proprio compito e termini il programma: nel gergo di Rust questa operazione è denominata `panic`.

2.3 Strutture dati

Un primo esempio di struttura dati messa a disposizione da Rust è il vettore: una collezione di elementi dello stesso tipo ai quali è possibile accedere tramite indice.

```
let vector = [1, 2, 3, 4, 5];
```

Un esempio di collezioni eterogenee sono le tuple, o, più in linea generale, n-uple: queste sono collezioni di valori di diverso tipo, affini alle tuple presenti in Scala~[9] e Haskell~[10], sono facilmente scomponibili per nome o per indice.

```
let tup = (15, "second");
let (first,second) = tup;
println!("{} {}", tup.0, second); //Printst 15 second
```

Vi sono casi invece in cui è necessario avere un tipo di dato che possa assumere una serie di possibili valori, è il caso delle enumerazioni.

```
enum Option<T> {
    Some(T),
    None,
}
```

`T` indica un qualunque tipo di dato; questa sopra è la più comune forma di enumerazione, una variabile di tipo `Option` può assumere due valori: `Some` oppure `None`: il primo indica la presenza di un valore all'interno della variabile mentre il secondo ne indica l'assenza. Il valore `None` è la scelta fatta dagli ideatori di Rust per sopperire alla mancanza del valore "null".
Un'ultima struttura dati messa a disposizione da Rust è la `struct`: questa permette di raggruppare all'interno di un tipo di dato variabili diverse che hanno qualcosa in comune:

```
struct Song{
    title : String,
```

```
    artist : String,
    duration : i32
}

let a = Song{
title : String::from("21st Century Schizoid Man"),
artist: String::from("King Crimson"),
duration: 511,
};
```

Le struct possono essere considerate come oggetti di linguaggi OOP che non sono dotati di comportamento ma solamente di attributi. E' possibile dotare le struct di un comportamento, in particolare è possibile implementare dei metodi:

```
impl Song{
    fn print_artist(&self) -> String{
        println!("{}", self.artist);
    }
}

a.print_artist();
```

in questo caso `&self` indica che questo metodo va chiamato su un dato di tipo Song.
Oltre ai metodi, all'interno dei blocchi `impl` è possibile definire delle "funzioni associate": queste hanno la sintassi identica a quella dei metodi ma non accettano `self` come parametro.

```
impl Song{
    fn print_artist(&self){
        println!("{}", self.artist);
    }
    fn print_artist_bis(par: Song) -> String{
        par.artist
    }
}
a.print_artist();
let b = Song::print_artist_bis(a);
```

Vengono chiamate funzioni associate in quanto sono associate alla struct stessa e non ad una sua istanza; vengono principalmente utilizzate per costruttori. Al contrario dei metodi, le funzioni associate vengono richiamate con il nome della struct seguita da "`::`" e infine il nome della funzione.

2.4 Controllo di flusso

Le forme più comuni di controllo all'interno di Rust sono i cicli e il costrutto `if`:

```
if number == 3 {
    //true condition
}else{
    //false condition
}
```

Come in Go~[11], le condizioni non sono racchiuse all'interno di parentesi, mentre sono necessarie per racchiudere i blocchi di istruzioni.
Rust mette a disposizione tre cicli distinti: `loop`, `while`, `for`. Si supponga uno scenario nel quale si deve incrementare di un'unità una variabile fino a quando non arriva a 10.

```
let mut counter = 0;
loop{
    counter += 1;
    if counter == 10{
        break;
    }
};
```

Il ciclo `while` non è altro che una versione più compatta di quello sopra citato:

```
let mut counter = 0;
while counter < 10 {
    counter += 1;
}
```

Se si conosce il numero di volte per le quali si vuole percorrere il ciclo, allora è opportuno utilizzare il costrutto `for`.

```
let mut counter = 0;
    for number in (1..10) {
        counter += 1;
    }
```

2.5 Macro

Una macro è una procedura simile ad una funzione che permette di racchiudere al suo interno codice richiamabile tramite il nome della macro. Vengono utilizzate per coprire le casistiche nelle quali una funzione non riesce a svolgere il compito necessario, come nel caso delle funzioni variadiche. Supponendo di voler realizzare una funzione che permetta di creare un vettore, è necessario supportare un numero indefinito di parametri: questa operazione non è supportata in Rust, non esistono funzioni variadiche, se non scritte in linguaggio esterno e poi utilizzate in scope "unsafe"; a questo scopo esistono le macro, ogni progetto ne utilizza qualcuna, a partire da una semplice println! fino ad arrivare a macro create ad hoc. Una macro è chiamata tramite il suo nome identificativo seguito da un punto esclamativo.

```
println!("Useless print");".
```

L'implementazione di una macro in Rust è più complessa rispetto alla realizzazione della stessa in un linguaggio come C, tuttavia i vantaggi sono numerosi; sono realizzate seguendo il modello per cui si ha un "matcher" e un "transcriber"~[12]:

```
macro_rules! myMacro{
    //rule 1
    ($name:expr) => {
        println!("This is my macro, {}",$name);
    };
    //rule 2
    ($name:stm) => {
        println!("This is my statement, {}",$name);
    };
}
```

dove "$name" è il matcher, mentre "expr" è il transcriber. Una volta invocata, la macro sopra controlla una ad una le proprie regole fino a trovare quella da attuare in base al parametro ricevuto in input. Vi sono più possibili valori di transcriber, ovvero parametri che la macro si aspetta, ed è possibile scegliere tra i seguenti:

- item: una funzione, una struct, etc;
- block: un blocco di istruzioni;
- stmt: uno stataement;
- pat: un pattern;

- ty: un tipo;
- ident: un identificatore;
- path: un percorso;

Questo modello porta giovamento in particolar modo dal momento in cui viene evitato il classico errore delle macro scritte in C per cui la traduzione avviene tramite una semplice sostituzione del corpo della macro che può portare ad errori di valutazione nelle precedenze degli operatori; in Rust questo problema è risolto grazie al fatto che l'argomento viene trattato come un unico valore.
Un altro problema risolto dall'utilizzo di questo modello di macro è quello dello shadowing di una variabile passata come argomento ad una macro che al suo interno contiene un'altra variabile con il medesimo nome: al momento dell'espansione del corpo della macro, la variabile esterna viene nascosta da quella interna, non generando il risultato corretto. Il problema viene risolto grazie alle "hygienic macro"~[12]: ogni macro viene espansa in un contesto a parte mentre ogni variabile viene associata al contesto in cui è stata creata, in questo modo anche se due variabili hanno lo stesso identificativo, il contesto sarà diverso e non vi saranno conflitti.

2.6 Pattern matching e destructuring declarations

Tony Hoare, informatico britannico ha definito la sua invenzione del valore `null` come "a bilion dollar mistake"[1]: Gli ideatori di Rust, concordando con Hoare hanno deciso di seguire questa filosofia non implementando il valore `null`; tuttavia è comunque necessario indicare l'assenza di valore.
Prendendo ad esempio il caso di una funzione che deve estrarre qualcosa da un file: questa operazione può fallire o riuscire; Rust mette a disposizione la funzione unwrap(): se l'operazione fallisce manda in panico il programma mentre se invece l'operazione ha successo essa restituisce il valore risultante. Per quanto comodo, non è una buona pratica di programmazione, sarebbe meglio distinguere e gestire i due casi; Rust utilizza il tipo `Result` $\langle T, E \rangle$: un enumerazione contentente due valori: successo oppure fallimento. Allo stesso scopo esiste il tipo `Option`, che ha sintassi simile svolge lo stesso compito degli `Optional` di Java e del `Maybe` di Haskell. Entrambi questi tipi vengono valutati tramite pattern matching, in particolare tramite il costruto "match", che è affine a quella implementata su Scala:

[1] `https://qconlondon.com/london-2009/qconlondon.com/london-2009/presentation/Null%2bReferences_%2bThe%2bBillion%2bDollar%2bMistake.html`

```
let optional : Option<i32> = None;
match optional {
    Some(_) => (println!("There is something here!")),
    None => (println!("There isn't anything")),
}
```

Match necessita di una casistica di default che copra tutti i casi rimanenti non coperti, indicata con l'underscore.
Un altro modo per valutare questi tipi di dato, in maniera meno dettagliata ma più veloce è il costrutto "if let", presente anche in Swift:

```
  let optional = None;
  if let Some(20) = optional{
    ()
  }else
  if let None = optional{
  println!("Empty!")
 }
```

Nel caso in cui si volesse fallire e mandare il programma in panico in caso di errore, Rust mette a disposizione l'operatore `?`. Posto alla fine della linea di codice che utilizza il costrutto Result, il punto di domanda restituisce il risultato se è presente, altrimenti lancia un errore.

Capitolo 3

Gestione della memoria

Esistono due approcci possibili al problema della memory-safety o più in generale alla gestione della memoria: il primo prevede un coinvolgimento in prima persona dello sviluppatore che si occupa direttamente, tramite codice, degli accessi in memoria e di assicurarsi che non vi siano problemi inerenti ad accessi ad aree di memoria indesiderate, è il caso ad esempio del linguaggio C; questo approccio garantisce buone performance ma al tempo stesso introduce una buona percentuale di errori dovuti alla natura umana dello sviluppatore che, per quanto esperto, tende a commettere errori. Il secondo approccio prevede che sia il garbage collector a gestire tutto ciò che concerne la memoria, lo sviluppatore non si deve occupare di liberare aree di memoria occupate o di evitare di liberarle più di una volta, in quanto è la macchina ad occuparsene; questo approccio garantisce una semplificazione del lavoro del programmatore, introducendo tuttavia dei costi in termini di performance dovuti alla presenza del garbage collector. Gli ideatori di Rust hanno deciso di non servirsi dell'utilizzo di un garbage collector in quanto le performance sono un aspetto importante di questo linguaggio, allo stesso modo lo sviluppatore non ha pieno controllo della memoria, di questa si occupa il compilatore, in modo particolare un suo componente denominato borrow checker, secondo la regola per cui dal momento in cui il codice viene compilato, non vi devono essere problemi di accessi illegali alla memoria. Nel dettaglio il compilatore segue tre regole fondamentali: ownership, borrowing rule e lifetimes.

3.1 Ownership

Avendo delegato al compilatore il compito di garantire la sicurezza della memoria, questo lo farà restringendo lo spazio di libertà dello sviluppatore intercettan-

do in anticipo pattern potenzialmente dannosi. Il principio di ownership si può suddividere a sua volte in tre regole:

1. ogni valore è contenuto all'interno di una variabile definita come "owner";
2. per ogni valore può esserci solo e solamente un owner alla volta;
3. quando l'owner esce dallo scope, il valore di sua proprietà viene rilasciato.

Partendo dal punto primo: ogni variabile è contenitrice e proprietaria di un valore; questo valore le appartiene fino a quando non viene spostato in un altra variabile tramite shadowing o move, oppure fino a quando la variabile non esce dal suo scope e il valore viene rilasciato. Il proprietario è univoco, ad una variabile corrisponde uno e un solo proprietario in un determinato istante (punto due). Rust di default alloca gli elementi sullo stack, utilizza lo heap solo per tipi di dato di dimensione variabile. Prendendo come esempio due variabili intere delle quali la seconda viene eguagliata alla prima, quello che succede è che la seconda variabile diventa una copia della prima: due variabili e due valori diversi; questo perchè il tipo di dato intero ha una dimensione fissa e farne una copia non comporta costi aggiuntivi.

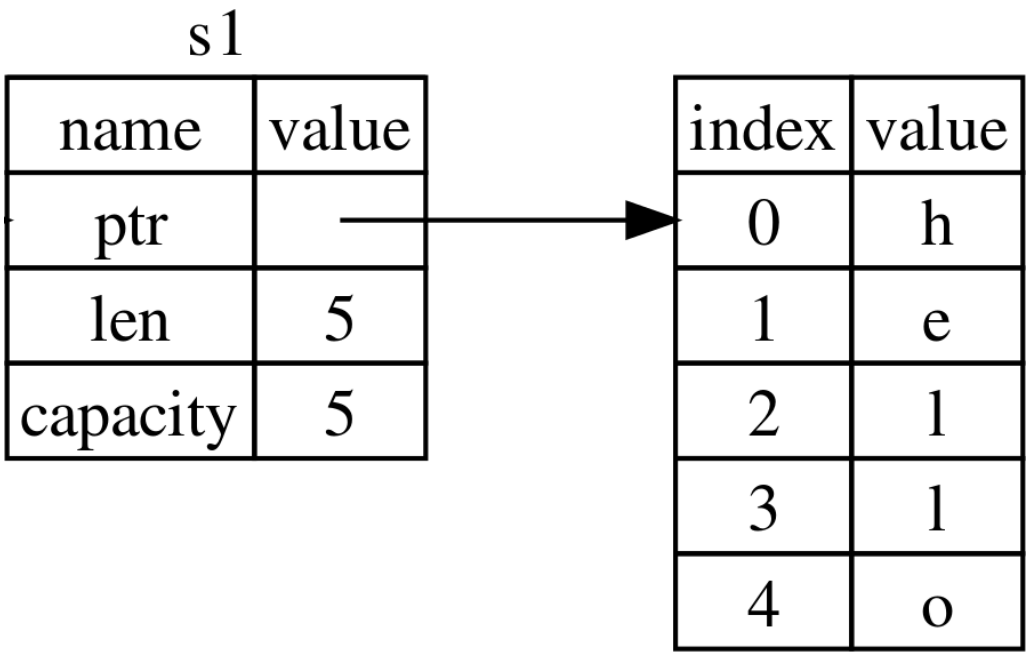

Figura 3.1: Struttura in memoria di una stringa, a sinistra il puntatore presente sullo stack, a destra la memorizzazione sullo heap

Diverso è il caso in cui entrano in gioco variabili di dimensione non fissa, come ad esempio le stringhe: salvate sullo heap mentre sullo stack è presente un puntatore con i dati necessari a recuperarne il contenuto.

```
let s1 = String::from("hello");
let new_var = s1;
```

In questo caso, `new_var` avrà, sullo stack, un puntatore alla struttura della stringa, contenuta nello heap, mentre `s1` verrà droppata; in gergo di Rust questa procedura è chiamata `move`, concretamente quello che avviene è uno spostamento dell'ownership. Cosa succede dunque alle variabili passate come argomenti di una funzione invocata? I valori contenuti all'interno delle variabili vengono mossi (da qui il termine move) all'interno della funzione, che diventa quindi owner di quei valori mentre le variabili esterne che prima li controllavano ne perdono la proprietà e vengono di conseguenza rilasciate.
Il terzo punto forza un pattern di buona programmazione usato anche in linguaggi come C++ ed è noto come RAII (Resource Acquisition Is Initialization) e svolge

il compito del garbage collector: quando una variabile esce dal suo scope, Rust invoca la funzione `drop` che si occupa di liberare l'area di memoria occupata da quella variabile.

3.2 Borrowing rule

E' sconveniente ogni qual volta venga invocata una funzione si perdano tutte le variabili passate come argomento alla funzione, per questo è stata introdotta la borrowing rule. Il concetto alla base di questa regola è proprio quello del prestare: quando una funzione viene invocata, invece della ownership degli argomenti, questi vengono prestati alla funzione che si occuperà di restituirli una volta terminato il suo procedimento; quando un riferimento esce dallo scope la variabile puntata non subisce alcun cambiamento. Il prestito avviene tramite il passaggio di un riferimento `&`, che non è altro che un riferimento all'area di memoria che contiene il valore che vogliamo prestare. A sua volta anche un riferimento può essere mutabile o immutabile.

```
fn my_fn(par1: &i32, par2: &mut i32) -> (){
    println!("I borrow and print two parameters");
    println!("{} and {}", par1, par2);
}

let mut a = "hello world";
let  b = &mut a;
*b = "goodbye world";
println!("{}",b);
```

Per evitare problemi di concorrenza o di inconsistenza dei dati il borrow checker controlla che sia rispettata la seguente regola:

- In un preciso istante è possibile avere uno ed un solo riferimento mutabile oppure un numero illimitato di riferimenti immutabili.

Questa semplice regola rimuove ogni possibilità di data race ed è fondamentale anche in caso di accesso concorrente~[13].
Sebbene la sintassi dei riferimenti sia analoga a quella dei puntatori in stile C vi sono due grosse differenze:

1. concettuale: un puntatore del linguaggio C punta ad un valore, un riferimento di Rust lo prende in prestito;
2. pratico: un puntatore del linguaggio C può accedere ad un valore nullo, un riferimento di Rust non può assumere valore nullo.

Per facilitare il lavoro dello sviluppatore Rust è provvista della “Deref coercion”: questa funzionalità fa si che lo sviluppatore non debba perdersi in operazioni di riferimento o deferenziazione (& o *) le quali vengono applicate autonomamente dal compilatore nei casi possibili, è ad esempio il caso in cui si fa la print di un riferimento ad una stringa, nel quale non c’è bisogno di specificare l’operatore di deferenziazione:

```
fn print_string(string : &str){
    println!("Printed: {}",string);
}
```

Allo stesso modo non è obbligatorio che un puntatore venga rilasciato alla fine del suo tempo di vita, questa operazione può essere anticipata tramite la funzione di libreria `drop`.

3.3 Lifetime

Introdotte le reference per risolvere un problema ne nasce un altro derivante da queste ultime: le ”dangling references”:

```
struct DanglingStruct ( &i32);
```

Quello sopra è l’esempio più banale di una possibile dangling reference; il compilatore necessita della certezza che non vi sia la possibilità di errori e questo codice non è memory-safe: ogni reference ha un tempo di vita (lifetime per l’appunto), in questo caso è necessario che il tempo di vita di quel riferimento sia maggiore o uguale a quello della struct stessa, in caso contrario il comportamento di questo codice è non definito e il compilatore lo rigetta. Per coprire questa mancanza sono stati introdotti i lifetimes espliciti:

```
struct DanglingStruct<'a> (&'a i32);
```

vengono indicati con un apostrofo e un nome, solitamente un singolo carattere; non sono altro che un’annotazione con la quale il programmatore specifica al compilatore che quel riferimento dovrà vivere e rimanere valido almeno tanto quanto la struttura onde evitare problemi di memoria.
Per quanto riguarda le funzioni, i lifetimes si suddividono in:

- input lifetimes: lifetime inerenti ai parametri di funzioni/methodi;
- output lifetimes: lifetime inerenti ai valori di ritorno di funzioni/metodi.

Le funzioni rappresentano la gran parte dei casi in cui è necessario l’inserimento di lifetimes, l’assegnazione di questi viene svolta, nelle casistiche più semplici, dal borrow checker stesso seguendo le “lifetimes elision rules”:

1. ad ogni parametro che è un riferimento viene assegnato un lifetime;
2. se è presente esattamente un input lifetime, questo viene asegnato a tutti gli output lifetime;
3. se sono presenti input lifetimes ma uno di questi è `self`, allora il suo lifetime è assegnato a tutti gli output lifetimes.

Queste tre regole non coprono tutte le casistiche, vi sono situazioni nelle quali il borrow checker non è in grado di assegnare i lifetimes e di conseguenza rifiuta il codice, in questo caso i lifetimes vanno specificati dallo sviluppatore.

3.4 Conclusioni sulla memory-safety

Per garantire la memory-safety, il compilatore di Rust preferisce rigettare programmi che potrebbero funzionare piuttosto che farne passare alcuni che potrebbero non funzionare, questo porta alcune volte a scrivere codice perfettamente funzionante ma che il compilatore rifiuta. A questo scopo Rust permette allo sviluppatore di utilizzare porzioni di codice che il compilatore rigetterebbe, purchè queste siano rinchiuse all'interno di uno scope dichiarato `unsafe`: è il caso ad esempio dei raw pointer, ovvero i veri e propri puntatori del C, così come, più in generale, ogni tipo di chiamata a codice C; le stesse funzioni di libreria di Rust sono costruite sopra a funzioni unsafe, benchè propriamente incapsulate.
È doveroso sottolineare che codice scritto all'interno di blocchi unsafe non è esente dal controllo del borrow checker o degli altri meccanismi di sicurezza di Rust, semplicemente permette l'utilizzo di quattro feature che sarebbero diversamente irrealizzabili:

- dereferenziare un raw pointer;
- accedere e modificare una variabile statica;
- implementare un trait unsafe (il cui concetto sarà introdotto in sezione 4.4)
- chiamare una funzione o un methodo unsafe;

In modo particolare l'ultima funzionalità può essere veramente dannosa, soprattutto se si introduce codice proveniente da altri linguaggi non sicuri, in quanto potrebbero verificarsi problemi di accessi ad aree di memoria non autorizzate; a questo scopo è stato recentemente introdotto Fidelius Charm~[14], una libreria scritta in C che mette a disposizione funzioni e macro che cercano di "proteggere" il più possibile la memoria all'interno di sistemi Linux prima dell'entrata in blocchi unsafe.

Con lo stesso scopo di Fidelius Charm ma con un approccio differente, SandCrust~[15] cerca di offrire maggiore sicurezza durante l'utilizzo di blocchi

di codice unsafe attraverso il sandboxing, l'isolamento di codice potenzialmente dannoso all'interno di un ambiente limitato, più nel dettaglio Sandcrust mira a separare il codice generato da blocchi unsafe in un processo differente.

A questo punto è spontaneo chiedersi se davvero Rust è memory-safe come affermato. Formalmente non esiste nessuna dimostrazione matematica dell'effettiva memory-safety del linguaggio, tuttavia è stato pubblicato RustBelt~[13], un modello semantico ideato con lo scopo di dimostrare la robustezza di una parte delle funzioni Rust che poggiano su codice unsafe; più nel dettaglio viene creato un λRust che è un nuovo linguaggio composto da solo e solamente le funzionalità di Rust che si vogliono provare memory safe. Questo sottoinsieme viene poi valutato per stabilire se una libreria che utilizza codice unsafe incapsulato in funzioni safe può essere definita safe.
La memory-safety abbinata alla velocità di Rust sta attirando negli ultimi anni molta attenzione, soprattutto in ambiti dove la prima è cruciale. Ad esempio nell'ambito astrofisico per la realizzazione di simulatori sono spesso preferiti linguaggi veloci come C, C++, o Fortran~[16], velocità ottenuta al costo di una maggiore complessità nella scrittura e nel testing. Rust consente di superare questa dicotomia dato che, in termini di prestazioni, Rust non è inferiore a linguaggi come Fortran o C.~[17].

Capitolo 4

Type System di Rust

4.1 Generici

Rust offre supporto ai tipi di dato generici, per convenzione vengono indicati con nomi brevi. Un esempio di utilizzo di generici è il tipo Result $\langle T, E \rangle$ descritto nei capitoli precedenti:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Utilizzati per limitare la duplicazione inutile di codice e migliorare l'organizzazione dello stesso, hanno un grande vantaggio rispetto ai tipi generici implementati in Java: mentre i secondi vengono implementati, compilati e eseguiti tramite type-erasure, che comporta un costo ulteriore a runtime, Rust effettua la "monomorfizzazione" del codice:

```
let intgr = Some(10);
let flt = Some(4.0);
```

il compilatore in questo caso capisce che Option $\langle T \rangle$ è stato utilizzato in due varianti : i32 e f64. A questo punto, esattamente come in C++, espande la definizione di Option$\langle T \rangle$ in Option$\langle i32 \rangle$ e Option$\langle i64 \rangle$ in modo che, al momento dell'utilizzo a runtime non sia necessario performare alcun tipo di modifica al codice e di conseguenza sia possibile evitare un costo aggiuntivo.

4.2 Trait

Un trait descrive un tipo delineandone le funzionalità esposte; sono analoghi alle interfacce di Java o C#, anche se, al contrario di queste, i trait non dispongono di ereditarietà.

```
pub struct MyStruct{
    pub number: i32,
}
pub trait ShowIt{
    fn show(&self) -> (){
        println!("Empty Show");
    }
}

impl ShowIt for MyStruct{
    fn show(&self) -> (){
        print!("{}",self.number);
    }
}
```

All'interno del trait per la funzione show è presente una "default implementation": grazie alla sua presenza qualunque tipo di dato voglia implementare il suddetto trait non necessita di ridefinire quella funzione. Così come in questo esempio è la struct myStruct ad implementare il trait ShowIt, si può dire che un trait accomuna tutti i tipi nei quali viene implementato; ad esempio ogni tipo di dato che può essere stampato tramite una print implementa il trait "Display". Non supportando l'ereditarietà, l'unico modo per mostrare il comportamento di un tipo di dato è tramite la keyword `impl`, implementando quello che viene definito come polimorfismo ad-hoc. È possibile dunque, richiedere come parametro in una funzione un tipo di dato qualunque purchè implementi un determinato trait.

```
pub fn toast(drnk1: impl Drink, drnk2: impl Drink ){
    println!("To us!")
}
```

4.2.1 Trait bound

L'esempio sopra può essere raffinato e ridefinito "trait bound": un trait bound permette di raggruppare ed identificare una lista di componenti dello stesso tipo

che implementano un determinato trait:

```
pub fn toast<T: Drinkable>(drnk1: T, drnk2: T){...}
```

C'è una piccola differenza tra i due esempi: mentre il primo permette di passare come argomenti alla funzione toast due elementi purché entrambi implementino il trait "Drinkable", la seconda necessita di due argomenti dello stesso tipo. Esistono varie forme di trait bound, legate alla chiarezza del codice: possono essere concatenati più di un bound tramite l'operatore "+":

```
pub fn toast<T: Drink + Light>(drnk1: T, drnk2: T){...}
```

Un trait bound può specificare anche che un valore di ritorno di una funzione implementi un determinato trait.

```
pub fn my_fnc<T : Drink>(par1: T) -> T{...}
```

4.3 Trait object

Rust è un linguaggio statico per cui, anche nel caso dei generici e trait, i tipi di dato sono noti al momento della compilazione, questo porta ad un risparmio di risorse altrimenti necessarie per un late binding. Vi sono casi tuttavia in cui si preferisce avere un peso in più in termini di risorse ma codice più chiaro e versatile. Si supponga di avere una funzione che prenda come argomento un qualunque dato implementi un determinato trait; Rust, a compile time, applicando la monomorfizzazione, crea una versione della funzione per ogni possibile tipo sul quale può essere invocata. È possibile evitare il procedimento di monomorfizzazione e imporre al compilatore di risolvere la versione della funzione a runtime tramite l'utilizzo dei trait object.

```
struct Foo;

trait Drink {
    fn toast(&self);
}
impl Drink for Foo {
    fn toast(&self) { println!("To us"); }
}

fn dynamic_dispatch(t: Box<dyn Drink>) {
}
```

`Box<dyn Drink>` è un trait object, dove `Box` è un contenitore che permette di allocare elementi di dimensione variabile sullo heap; `dyn` invece è una parola chiave con scopo esplicativo: non ha alcuna funzionalità se non indicare al programmatore la presenza di un trait object: a runtime sarà presente una sola versione di questa funzione. Questo risultato è ottenuto tramite "dynamic dispatch", che non è altro che quello che Java effettua con i generici, ovvero una type-erasure per poi valutare a runtime il tipo di dato esatto

4.4 Coherence and The orphan rule

Non tutti i trait possono essere trasformati in trait object, nello specifico possono essere trasformati in trait object solamente i trait object-safe.
Un trait è definito object safe se:

- il tipo di ritorno non è self;
- non ci sono parametri generici.

I trait sottostanno a proprietà ben precise che garantiscono la sicurezza del linguaggio e sono Coherence e Orphan Rule. Con coherence si intende l'impossibilità di avere più di un implementazione di un trait per un certo tipo: per quanto banale possa essere, l'assenza di questa regola porterebbe conflitti ininterrottamente all'interno del codice; in altri linguaggi sprovvisti di coherence come ad esempio Haskell, l'implementazione è scelta arbitrariamente. L'Orphan Rule è un ulteriore rinforzo alla coerenza:

- Non è possibile fornire un'implementazione di un trait per una struct se non si è proprietari del crate (verrà definito nella sezione 5.1) che definisce la struct o del crate che definisce il trait.

Questa regola può essere riassunta con la necessità che uno tra il trait o la struct sulla quale esso è implementato sia locale. Dalla sua origine l'orphan rule è stata messa in discussione per la troppa restrittività, in quanto ad esempio una situazione simile non è ammessa sebbene coerente:

```
impl<T> From<Foo> for Vec<i32>
```

dove Foo è un trait esterno e Vec è, ovviamente, un tipo esterno. In Gennaio 2019 è stata approvata l'RFC#2451[1] con la quale si permette l'implementazione di trait esterni per tipi esterni. La restrittività di Rust è continuamente messa in discussione e proposte di modifiche vengono frequentemente alla luce con lo scopo di semplificare il linguaggio e renderlo più agevole, pur mantenendo la rigidità che lo contraddistingue.

[1] `https://rust-lang.github.io/rfcs/2451-re-rebalancing-coherence.html`

4.5 Chiusure

Le chiusure sono un aspetto che Rust prende dalla programmazione funzionale; assomigliano alle lambda di altri linguaggi come Python mentre la sintassi è simile a quella di linguaggi come Smalltalk[18] o Ruby. Sono funzioni anonime memorizzabili all'interno di variabili.

```
let my_closure = |x| x+3;
```

Vengono utilizzate principalmente all'interno degli iteratori, nella programmazione concorrente e per limitare la duplicazione di codice.
Al contrario delle funzioni, le closures possono accedere alle variabili all'interno dello scope in cui sono definite; chiaramente questo comporta un overhead di memoria che non sempre è necessario pagare. Esistono tre modi in cui una closure può appropriarsi delle variabili all'interno dello scope, tali dipendono dal tipo della closure:

- FnOnce: tramite una move e di conseguenza prendendone l'ownership, le variabili dello scope vengono spostate nello scope interno della closure; una closure di questo tipo può prendere il controllo di una variabile una sola volta;
- FnMut: la closure prende in prestito (borrowing) lo scope esterno e può cambiare i valori delle variabili;
- Fn: la closure prende in prestito lo scope esterno senza modificarlo.

È possibile, come nell'esempio sopra, non specificare il tipo e il valore di ritorno della closure in quanto, nella maggior parte dei casi, il compilatore è in grado di dedurlo autonomamente; in caso non riesca, la compilazione fallisce.

4.6 Smart pointers

Rust eredita da C++, dove per primi sono stati introdotti, gli smart pointers: strutture dati che oltre ad agire come puntatori possiedono metadati e funzionalità aggiuntive; al contrario dei riferimenti, questi puntatori spesso sono proprietari dei dati ai quali puntano e vengono costruiti tramite struct. Due proprietà fondamentali dei puntatori in generale sono la deferenziazione e il drop. Rust si occupa autonomamente di effettuare il drop di elementi che esauriscono il loro tempo di vita uscendo dal loro scope, tuttavia, così come per la deferenziazione, è possibile, nel caso si utilizzano puntatori creati ad hoc, personalizzare queste due proprietà tramite i due rispettivi trait: `Deref` e `Drop`. Rust fornisce, oltre alla possibilità di realizzarli da zero, una serie di smart pointers:

- `Box`$\langle T \rangle$: Rust di default memorizza gli elementi sullo stack, Box è un contenitore di dimensione fissa che permette di allocare sullo heap elementi di dimensione non nota, come ad esempio elementi ricorsivi;
- `Rc`$\langle T \rangle$: acronimo per Reference Counting, tiene traccia di quanti riferimenti attivi vi sono ad un determinato elemento, di conseguenza è utilizzato in contesti in cui non è chiaro chi sia l'owner di quel determinato elemento; se il reference counting vale zero, allora l'elemento puntato può essere droppato.
- `RefCell`$\langle T \rangle$: smart pointer che forza il controllo del borrow checker a run time: può succedere che, per natura conservativa del compilatore, codice sicuro venga rifiutato perchè il borrow-checker non è in grado di stabilire la sicurezza del codice; tramite RefCell, è possibile bypassare questo problema.

4.6.1 Interior Mutability Pattern

Si supponga di avere una struct immutabile con all'interno due campi; per le regole di Rust, nessuno dei campi interni può essere modificato in quanto la struct stessa non è modificabile.

```
struct Dummy{var : i32}
let my_var = Dummy{ var : 1};
my_var.var = 2;
```

Questa restrizione pone dei grossi limiti, in quanto rendere tutta la struct mutabile espone tutti i campi interni a modifiche esterne e non è una buona pratica di programmazione.
L'interior mutability è un pattern che permette di modificare degli elementi anche quando esistono riferimenti ad essi. Un modo per applicare questi pattern è tramite RefCell$\langle T \rangle$.

```
struct Dummy{var : RefCell<Vec<i32>>}

impl Dummy{
    pub fn add(&self,value : i32) -> (){
        self.var.borrow_mut().push(value);
    }
}

let my_var = Dummy{ var : RefCell::new(vec![])};
my_var.add(3);
```

In questo esempio il vettore è stato avvolto da uno smart pointer: inizialmente viene acquisito un riferimento mutabile al contenuto dello smart pointer tramite `borrow_mut()`; successivamente viene inserito all'interno del vettore il nuovo elemento tramite `push`. Questa implementazione ha come pro il fatto che nonostante il vettore venga modificato, appare comunque immutabile dall'esterno; tuttavia, oltre ad un overhead di risorse dovuto al check a runtime, è necessario assicurarsi per lo stesso motivo che il codice scritto non abbia alcun difetto, in quanto non verrà rivelato ma causerà il panic del programma.
RefCell tiene traccia di quanti riferimenti mutabili e immutabili sono stati fatti all'elemento avvolto: come nella normale borrowing rule, è permesso avere un riferimento mutabile o infiniti immutabili.

Capitolo 5

Altri Aspetti

5.1 Struttura di un progetto Rust

Cargo è un build system per Rust, i suoi principali compiti sono quelli di organizzare l'ambiente di sviluppo, gestire le dipendenze e occuparsi della build del progetto. Un progetto Rust è suddiviso in packages e crates: un package può contenere uno o più crate e contiene un file Cargo.toml, che è un descrittore del progetto ed è strutturato nel seguente modo:

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Manuele Pasini <manuele.pasini@studio.unibo.it>"]
edition = "2018"

[dependencies]
rand = "0.6.5"
```

Le dipendenze sono riferimenti a crate esistenti che vengono importati nel progetto; Cargo effettua il download da crates.io[1], che è un registro di tutti i package di Rust pubblicati dalla sua comunità. Un crate può essere binario (.src) o una libreria (generalmente `lib.rs`); un package può contenere al massimo un file di libreria, generalmente denominato "lib.rs", mentre non è invece presente un limite sul numero di file binari. E' possibile utilizzare funzioni di altri crate purchè importati tramite la keyword `use` e inseriti nelle dependencies se si tratta di crate non presenti in locale. A loro volta i crate possono essere suddivisi in moduli: si può dire che sono quello che più assomiglia alla suddivisione in package di Java,

[1] `https://crates.io/`

vengono utilizzati per garantire leggibilità e suddivisione del codice e soprattutto per controllare la visibilità del codice; viene dunque a formarsi una struttura ad albero. E' possibile legare più crate tra di loro a formare una workspace: si supponga di voler creare una workspace che contenga al suo interno due crate, `first-crate` e `second-crate`; una volta codificati questi due crate è possibile legarli tra loro tramite il file `Cargo.toml`, specificando i membri della workspace.

```
[workspace]

members = [
    "first-crate",
    "second-crate",
]
```

Prendendo come ide Visual Studio Code[2], la workspace sopra verrà rappresentata nel modo seguente:

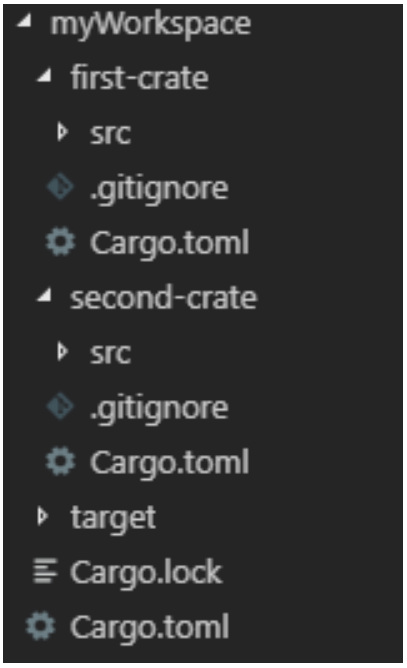

Figura 5.1: Esempio di workspace

[2] `https://code.visualstudio.com/`

Al contrario di linguaggi come C, dove le procedure per produrre un eseguibile vanno svolte manualmente dallo sviluppatore, come linkare i moduli o scrivere il Makefile, Cargo si occupa autonomamente di queste cose; lo sviluppatore si occupa solamente di interagire con Cargo tramite riga di comando, particolare menzione per il comando `cargo build`, che genera l'eseguibile: sono disponibili due profili di build, quello di "dev" e quello di "release", che differiscono per il livello di ottimizzazione.

Attualmente Rust non è dotato di un IDE di riferimento, anche se il supporto è crescente; IDE come Vim, Emacs, VSCode ed Eclipse supportano il linguaggio, a denotare un interesse crescente. In particolare il refactoring del codice, è un processo da svolgere a mano da parte dello sviluppatore; probabilmente si tratta di una funzionalità che verrà sviluppata man mano e se il linguaggio continuerà a progredire, tanto che è già stata abbozzata e pubblicata una prima versione di quello che potrebbe essere un tool per la rifattorizzazione del codice~[19], che si concentra in modo particolare sul rinominare variabili, funzioni, etc; inlining di variabili e un miglioramento inerente alla esplicitazione dei lifetimes, il tutto basato su uno dei primi tool per la rifattorizzazione: "Smalltalk Refactoring Browser".

5.1.1 Testing

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."~[20] Oltre al build system, Rust offre supporto a livello di linguaggio per il testing. Il processo di testing è diviso in due categorie: unit test e integration test; i primi sono test interni ai vari moduli, nei quali si testa il corretto funzionamento di procedure private e all'interno dei singoli moduli; i test d'integrazione d'altro canto utilizzano il codice come se venisse chiamato da fonti esterne. Il codice può essere testato tramite comando `cargo test`, attraverso il quale Cargo eseguirà solamente le funzioni etichettate tramite `#[cfg(test)]`; una volta eseguite effettuerà un resconto con il numero di funzioni eseguite, quante hanno fallito e quante invece hanno avuto successo.

```
running 6 tests
test tests::greater_than_100 ... ok
test tests::greeting_contains_name ... ok
test tests::it_adds_two ... FAILED
test tests::it_works ... ok
test tests::larger_can_hold_smaller ... ok
```

```
test tests::smaller_cannot_hold_larger ... ok
test result: FAILED. 5 passed; 1 failed;
```

Al contrario di altri linguaggi, Rust consente il testing.

Seguendo la semantica strutturale di Rust, gli integration test vanno inseriti in una directory a parte, allo stesso livello della cartella src. All'interno del file di test non è necessario etichettare le funzioni come per gli unit test, in quanto Cargo riconosce la cartella come cartella di test e la sfoglia solo in caso di comando specifico. I test vengono svolti prendendo la funzione che si vuole testare, eseguita con certi parametri e poi confrontata tramite un assert (un'altra macro) con il valore che dovrebbe restituire nel caso in cui la funzione fosse corretta.
Di default Rust esegue i test in thread paralleli ; è possibile tuttavia specificarre a Cargo il numero di thread che si vogliono utilizzare, conveniente nel caso in cui le funzioni da testare accedano a risorse condivise.

5.2 Concorrenza in Rust

La concorrenza è un aspetto importante in un linguaggio di programmazione, soprattutto quando si parla di un linguaggio che fa della gestione della memoria (in termini di sicurezza) e dell'efficienza i suoi punti chiave. Essendo un linguaggio di basso livello, Rust ha scelto di implementare, nella libreria standard, l' 1:1 threading, ovvero un kernel thread per ogni thread dell'utente. Esistono tuttavia crates che forniscono implementazioni di thread m:n, per cui vi è un unico kernel thread sul quale poggiano più thread utente. Un thread può essere creato tramite la funzione "thread::spawn", che necessita come argomento una closure: un primo problema incontrato è quello di passare una variabile al thread appena creato; di default la closure prende in prestito la variabile, tuttavia il borrow-checker non riesce a capire se quel riferimento ha un tempo di vita maggiore o uguale a quello della closure, per questo motivo, è necessario che il thread prenda l'ownership della variabile passata, tramite la parola chiave "move". La funzione spawn restituisce una variabile di tipo JoinHandle attraverso la quale è possibile attendere la terminazione di tutti i thread.

```
fn main() {
    let my_vec = vec![1, 2, 3, 4];

    let handle = thread::spawn(move || {
        println!("moved value: {}", v);
    });
```

```
    handle.join();
}
```

In questo ambito, Rust è un linguaggio molto più somigliante a Go ed Erlang~[21] piuttosto che a C o C++, tant'è che condivide il mantra "do not communicate by sharing memory; instead, share memory by communicating."[3]; a questo scopo Rust mette a disposizione un canale di tipo multi produttore e singolo consumatore (che può diventare multi tramite clone). La ricezione di un messaggio può avvenire in due modi: bloccando il thread corrente in attesa di un messaggio, oppure tramite un controllo istantaneo che restituisce un tipo Result. Anche in questo caso, l'ownership per quanto renda più difficile la comprensione del modo in cui Rust gestisce i thread e per quanto più complicato possa essere scrivere codice che la rispetti, permette allo sviluppatore di non preoccuparsi di errori altrimenti frequenti, come ad esempio l'utilizzo di una variabile dopo averla passata nel canale. Un altro metodo di comunicazione tra thread proposto da Rust è quello che coinvolge i mutex, primitive per la condivisone di memoria. In questo caso la presenza del borrow-checker facilita l'utilizzo dei mutex in quanto i problemi più frequenti di questo metodo sono l'utilizzo di dati prima di aver preso la mutua esclusione o il mancato rilascio della mutua esclusione al termine dell'utilizzo; questi errori sono tutti rilevati a compile time.
Questo approccio genera un problema dal momento in cui la variabile mutex deve essere condivisa tra più thread, in quanto l'ownership di quest'ultima va data ad un singolo thread ed ogni tentativo dei rimanenti di accederci porta ad un errore di compilazione. Allo stesso modo, lo smart pointer Rc, che poteva essere una soluzione, non è thread safe e dunque non utilizzabile. Rust fornisce un implementazione thread-safe di questo puntatore, denominato Arc$\langle T \rangle$. A questo punto, con la variabile mutex che fornisce, esattamente come RefCell, l'interior mutability, è possibile far si che più thread accedano e modifichino la stessa variabile condivisa, nell'esempio sottostante 10 thread aumentano, ciascuno, la variabile counter di un'unità. La scelta di avere due differenti reference counter, uno thread-safe e l'altro no è dovuta al fatto che il primo necessita chiaramente di un overhead di risorse che, in caso di programmazione non concorrente, non è necessario pagare, dunque è stato deciso di scinderli in due tipi di dato diversi.

```
    let counter = Arc::new(Mutex::new(0));
    let mut tjreads = vec![];
    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
```

[3] https://blog.golang.org/share-memory-by-communicating

```
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        threads.push(handle);
    }

    for handle in threads {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
```

I meccanismi proposti da Rust per garantire la memory-safety agevolano alcuni aspetti e problemi presenti quando si parla di programmazione concorrente, tuttavia è necessario capire se questi meccanismi comportano un costo aggiuntivo soprattutto in termini di prestazioni; due studi in particolare hanno mostrato, in una simulazione che non replica integralmente una situazione reale, le prestazioni di Rust in ambiente concorrente comparate con quelle di C[4] e con quelle Go[5]. In entrambi i casi è possibile vedere che Rust non introduce un elevato ritardo di esecuzione rispetto agli altri due, benchè comunque in entrambi gli studi è presente una menzione particolare alla difficoltà nell'imparare questo linguaggio.

[4] https://ehnree.github.io/documents/papers/rustvsc.pdf

[5] http://cs242.stanford.edu/f17/assets/projects/2017/gsfisher-chrisyeh.pdf

Capitolo 6

Conclusioni

6.1 Rust è un linguaggio di programmazione ad oggetti?

Una domanda che nasce spontanea una volta conosciute ed approfondite tutte le funzionalità di Rust è proprio quella che chiede se è possibile classificare questo linguaggio come linguaggio di programmazione ad oggetti. Una definizione di linguaggio ad oggetti è la seguente: ”Un linguaggio ad oggetti è composto da oggetti. Un oggetto racchiude in se dei dati, chiamati attributi, e delle procedure che operano su esso, definite metodi.” Seguendo questa definizione, Rust può essere etichettato come linguaggio OOP, in quanto al posto degli oggetti conosciuti in Java possiede le struct, che al suo interno hanno attributi, e le procedure non sono altro che i blocchi ”impl” all'interno dei quali vengono definiti i metodi di una determinata struct.
Se invece si fa riferimento alla definizione che, in un linguaggio ad oggetti è in grado di implementare

- incapsulamento;
- polimorfismo;
- ereditarietà

allora Rust non rientra in questa categoria. L'incapsulamento è una proprietà di cui Rust gode, basti pensare alla visibilità dei metodi di una struct, questi possono essere privati o pubblici, che è quello che l'incapsulamento richiede; il polimorfismo allo stesso modo è garantito dalla presenza di tipi di dato generici e trait, più precisamente quello fornito da Rust è un polimorfismo parametrico, secondo cui una funzione può essere scritta in modo generico da far si che si possa comportare allo stesso modo senza dipendere da un tipo fisso. Rust non gode di ereditarietà sebbene fornisca alternative come i trait, che possono

simulare una condivisione di codice tra struct diverse. E' possibile dunque affermare che Rust non è un linguaggio di programmazione ad oggetti, sebbene ne erediti alcuni aspetti e ne emuli alcune caratteristiche.

Il punto di forza di Rust risiede nella memory-safety senza dover pagare il costo di un garbage collector. I concetti introdotti in termini di gestione e sicurezza della memoria sono innovativi ed efficaci; dall'altro lato la curva di apprendimento di questo linguaggio è decisamente ripida, spesso ci si scontra con problematiche che in altri linguaggi non si sono mai dovute affrontare perchè gestite dai compilatori. Questa innovazione ha portato e probabilmente continuerà a portare in futuro ad adattamenti da parte di altri linguaggi di programmazione, ad esempio è stato lanciata l'idea che Scala potesse importare qualcosa da Rust~[22] tramite l'implementazione di un concetto di borrowing. D'altronde anche Rust stesso è ispirato da diversi linguaggi di programmazione, dai quali si è cercato di prendere qualunque cosa potesse garantire a Rust di raggiungere il proprio goal: efficienza e memory-safety. A quest'ultima si aggiunge, tra le caratteristiche di Rust, la facilità con cui è possibile implementare meccanismi di software fault isolation o controlli di flusso dei dati all'interno del software~[23].
Inevitabilmente, per la complessità del linguaggio, vi sono problematiche inerenti alla realizzazione di determinati progetti, come ad esempio quelle rinvenute nella realizzazione di Tock~[24], un sistema operativo embedded, problematiche che riguardano la rigidità del compilatore che, per scelta progettuale, preferisce rigettare codice che non riesce a certificare sicuro piuttosto che farne passare di insicuro.
Allo stesso tempo però, Rust ha ricevuto numerosi elogi in qualità di linguaggio di basso livello per la realizzazione di garbage collector~[25] e per l'utilizzo nello sviluppo di un kernel~[26]. Nonostante le difficoltà di apprendimento Rust negli ultimi anni sta guadagnano consensi all'interno della comunità informatica, tanto da classificarsi primo nel 2019 per il quarto anno di fila nella classifica di StackOverflow dei linguaggi di programmazione più amati [1]. Allo stesso modo il linguaggio sta guadagnando popolarità anche nella comunità scientifica, che partecipa ed è fonte primaria dei cambiamenti e nuove proposte mirate al miglioramento di Rust. Considerando il palcoscenico di nicchia al quale è rivolto Rust, sarà interessante vedere se riuscirà a diffondersi tra tutti gli altri linguaggi di basso livello già esistenti e dominanti.

[1] https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

Capitolo 7

Ringraziamenti

Al prof. Viroli e Ing. Pianini per l'opportunità datami.

Alla mia famiglia, a Francangelo, Tiziana, Mirco e Mattia, che non mi hanno mai fatto mancare niente, hanno sempre dato tutto e mi avrebbero dato ancora di più se ce ne fosse stato bisogno, tutto perchè io potessi arrivare fin qui.

Ad Alessio, a tutte le persone che mi sono state accanto in questi anni, in particolar modo a Dendo e Giacomo, che hanno reso questa esperienza indimenticabile e senza i quali non sarei qui oggi.

Bibliografia

[1] Arthur Azevedo de Amorim, Cătălin HriŢcu, and Benjamin C. Pierce. The meaning of memory safety. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 79–105, Cham, 2018. Springer International Publishing.

[2] Brian W. Kernighan. *C Programming Language, 2nd Edition*. Prentice Hall, apr 1988.

[3] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, jan 2018.

[4] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the servo web browser engine using rust. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*. ACM Press, 2016.

[5] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[6] Nicholas D. Matsakis and Felix S. Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, October 2014.

[7] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, may 2013.

[8] Tony Hetherington. An introduction to the extended pascal language. *ACM SIGPLAN Notices*, 28(11):42–51, November 1993.

[9] Martin Odersky. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Inc, jan 2011.

[10] Bryan O'Sullivan. *Real World Haskell*. O'Reilly Media, dec 2008.

[11] Alan A. A. Donovan. *The Go Programming Language (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, nov 2015.

[12] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*. ACM Press, 1987.

[13] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, December 2017.

[14] Hussain M. J. Almohri and David Evans. Fidelius charm. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy - CODASPY '18*. ACM Press, 2018.

[15] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems - PLOS'17*. ACM Press, 2017.

[16] Walter S. S. Brainerd. *Guide to Fortran 2003 Programming*. Springer, sep 2014.

[17] Sergi Blanco-Cuaresma and Emeline Bolmont. What can the programming language rust do for astrophysics? *Proceedings of the International Astronomical Union*, 12(S325):341–344, October 2016.

[18] Alec Sharp. *Smalltalk by Example: The Developer's Guide*. McGraw-Hill, apr 1997.

[19] Garming Sam, Nick Cameron, and Alex Potanin. Automated refactoring of rust programs. In *Proceedings of the Australasian Computer Science Week Multiconference on - ACSW '17*. ACM Press, 2017.

[20] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[21] Francesco Cesarini. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, jun 2009.

[22] Leo Osvald and Tiark Rompf. Rust-like borrowing with 2nd-class values (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala - SCALA 2017*. ACM Press, 2017.

[23] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS '17*. ACM Press, 2017.

[24] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems - PLOS '15*. ACM Press, 2015.

[25] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management - ISMM 2016*. ACM Press, 2016.

[26] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems - APSys '17*. ACM Press, 2017.