Corso di Laurea in Informatica

TESI DI LAUREA

Oltre ownership e borrowing: Analisi, progetto e implementazione di smart pointers in Rust

Relatore:
Andrea Corradini

Correlatore:
Gian-Luigi Ferrari

Candidato:
Tommaso Di Vito

ANNO ACCADEMICO 2022/2023

Indice

Introduzione

Rust è un linguaggio di programmazione multi-paradigma, fortemente tipato e progettato per soddisfare i requisiti tipici della programmazione di sistema, cioè per la scrittura di software come kernels di sistemi operativi, firmwares, drivers e simili. La programmazione di sistema richiede, tra le altre cose, efficienza di esecuzione e la memory safety (assenza di errori di accesso alla memoria) in modo da assicurare l'assenza di problemi di sicurezza (come buffer overflow e memory leakage). Il linguaggio riprende anche concetti della programmazione funzionale e della programmazione a orientata agli oggetti supportando le chiusure, strutture e metodi, i generics per lavorare con tipi parametrici e i traits sia per supportare l'overloading degli operatori, sia per definire il comportamento dei metodi di una struttura in modo analogo alle interfacce del linguaggio Java, ma senza ereditarietà. Rust si distingue per le caratteristiche di efficienza, velocità di esecuzione e gestione safe della memoria. Rust prevede sia un controllo a basso livello della memoria, sia delle regole rigide per l'allocazione e la deallocazione dei dati sullo stack e nello heap senza l'utilizzo di un garbage collector. Il linguaggio adotta un type system basato su ownership e borrowing che con il solo utilizzo di analisi statica del codice garantisce che un programma che compila correttamente non può presentare gli errori tipici di gestione della memoria (come ad esempio use after free, buffer overflow e race conditions). Quindi il compilatore di Rust adotta un approccio di tipo 'fail-safe', rifiutando ogni programma di cui non si riesce a dimostrare l'assenza di errori mediante l'analisi statica del codice.

L'obiettivo di questa tesi è quello di studiare ed elaborare tecniche di programmazione Rust per strutture dati che permettono di superare la rigidità delle regole del linguaggio, pur continuando a garantire la memory safety. In particolare la tesi affronta le problematiche relative all'utilizzo degli smart pointers per risolvere problemi non risolvibili o non risolvibili in maniera efficiente in Rust a causa delle regole troppo stringenti.

Le regole rigide di Rust rendono impossibile una diretta manipolazione di strutture dati circolari. Per risolvere questo problema, il linguaggio offre alcune funzionalità aggiuntive che permettono di violare alcune delle regole, tollerando l'utilizzo di alcune operazioni di cui non è possibile verificare la correttezza. La prima parte della tesi si concentra sui meccanismi offerti dal linguaggio per la

realizzazione di smart pointers e esamina lo smart pointer GhostCell [33]. GhostCell definisce una API per l'implementazione di tipi ricorsivi e strutture dati cicliche che garantisce il rispetto delle regole di Rust senza aggiungere costi a tempo di esecuzione. La tesi riprende ed estende in modo originale alcune delle idee di GhostCell per l'implementazione di uno smart pointer per operare su grafi in maniera più efficiente rispetto a quanto fatto da quest'ultimo.

Il contributo originale della tesi è la progettazione e realizzazione di GenerationalGraph. GenerationalGraph è uno smart pointer di Rust che permette di operare su grafi suddivisi in generazioni in modo efficiente utilizzando esclusivamente codice safe (cioè codice soggetto a tutte le regole del linguaggio). *Di seguito utilizzeremo l'acronimo G^2 per indicare lo smart pointer GenerationalGraph*. G^2 è stato ottenuto unendo parte dei concetti introdotti da GhostCell con il meccanismo delle chiusure e dei lifetimes offerto dal linguaggio in modo differente e migliore rispetto a quanto fatto da GhostCell. In particolare G^2 consente di collegare nodi tra generazioni diverse in sicurezza sia in ambito single-thread, sia in ambito multi-thread senza l'utilizzo di un garbage collector, cosa che GhostCell non permette in alcune situazioni o permette in modo limitato. G^2 può essere utilizzato per l'implementazione dei vari algoritmi su grafi con due modalità diverse. Può essere usato esclusivamente tramite codice safe per contesti dove è necessaria la sicurezza dell'implementazione prima delle performance, pagando alcuni costi aggiuntivi dovuti alla gestione dei nodi. Ma è anche possibile estendere G^2 tramite codice unsafe se non si vogliono tollerare quei costi aggiuntivi, seguendo alcune regole in modo da garantire la correttezza dell'estensione.

Struttura della tesi

Nel primo capitolo introduciamo i concetti generali di Rust, concentrandoci sugli aspetti offerti dal linguaggio per garantire la memory safety. Iniziamo presentando alcuni aspetti esclusivi del linguaggio come il meccanismo di ownership e borrowing e il meccanismo di rilascio della memoria imposto dal linguaggio. Successivamente vengono presentati i tipi di dati principali presenti nel linguaggio, analizzando anche l'impatto delle regole imposte da Rust su alcuni di essi. Infine vengono presentati sia il funzionamento del multi-threading in Rust con le relative regole da seguire per poterlo effettuare in sicurezza, sia le funzionalità unsafe offerte dal linguaggio per effettuare operazioni non normalmente permesse. Nel corso del capitolo sono presentati vari esempi deliberatamente errati di utilizzo delle varie funzionalità accompagnati dal relativo messaggio di errore,

eventualmente seguiti da un possibile esempio di utilizzo corretto.

Nel secondo capitolo approfondiamo il meccanismo dei lifetimes per la verifica degli utilizzi dei riferimenti creati attraverso il meccanismo di borrowing offerto dal linguaggio per dare accesso temporaneo al valore di una variabile. Inoltre approfondiremo il concetto di varianza introdotto dal linguaggio per permettere una flessibilità maggiore nell'utilizzo dei lifetimes, consentendo di usare riferimenti che hanno una durata diversa, in contesti dove il loro utilizzo non può portare a violare la memory safety causando la creazione di dangling references. Questo meccanismo può essere utilizzato anche in contesti diversi da quello per cui è stato progettato e quindi vengono approfonditi anche alcuni utilizzi più avanzati per la creazione di relazioni tra strutture non normalmente correlate a causa dell'utilizzo delle funzionalità unsafe.

Nel terzo capitolo approfondiamo il concetto di smart pointer per la gestione di una zona di memoria allocata principalmente nello heap, discutendo innanzitutto le loro origini e il loro scopo all'interno del linguaggio, visto che non sono una funzionalità esclusiva di Rust, ma provengono da altri linguaggi. Successivamente discutiamo la necessità del loro utilizzo per implementare tipi ricorsivi in Rust, presentando alcuni aspetti che il linguaggio offre per utilizzare più agevolmente questi tipi. Dopo aver introdotto i concetti fondamentali degli smart pointers in Rust, analizziamo gli smart pointers predefiniti più comunemente utilizzati sia per l'utilizzo in ambito single-thread, sia in ambito multi-thread.

Nel quarto capitolo analizziamo nel dettaglio GhostCell [33], lo smart pointer preso come caso di studio per l'implementazione di tipi ricorsivi, discutendo inizialmente le motivazioni che hanno spinto alla definizione di questo smart pointer, dato che ne esistono altri che permettono di implementare tipi ricorsivi. Successivamente analizzeremo nel dettaglio la sua implementazione in Rust, mostrando gli aspetti insoliti per uno smart pointer e mostreremo i vantaggi offerti da quest'ultimo mediante l'implementazione delle liste concatenate doppie in ambito single-thread e multi-thread sia con GhostCell, sia con gli smart pointer predefiniti del linguaggio. Dopo aver visto le implementazioni, le metteremo a confronto, discuteremo eventuali vantaggi e svantaggi dell'utilizzo di GhostCell rispetto a altri smart pointer, forniremo dei brevi benchmarks sulle varie implementazioni e infine discuteremo alcune possibili ottimizzazioni.

Nel quinto capitolo presentiamo il nuovo smart pointer GenerationalGraph (G^2) per creare e manipolare grafi in modo efficiente in Rust, in modo che sia possibile suddividere i grafi in 'generazioni'.

Iniziamo discutendo delle problematiche introdotte dall'utilizzo di GhostCell per la definizione di questo tipo di struttura ricorsiva e successivamente discutiamo nel dettaglio le scelte di implementazione per ogni struttura utilizzata e per ogni metodo definito per il nuovo smart pointer. Inoltre, mostriamo come sia possibile estendere la sua definizione per implementare i vari algoritmi su grafi tramite codice safe e eventualmente tramite codice unsafe. Dopo aver definito lo smart pointer per il caso single-thread, analizzeremo eventuali problematiche introdotte dal suo utilizzo in un contesto multi-thread. Per finire forniremo dei brevi benchmarks che mettono a confronto altri smart pointer già esistenti per la creazione e manipolazione di grafi (tra cui il nostro caso di studio) per mostrare le differenze di performance in termini di tempo.

1. Panoramica di Rust

Rust [25] è un linguaggio di programmazione multi-paradigma, fortemente tipato e progettato per soddisfare i requisiti tipici della programmazione di sistema, cioè per la scrittura di software come kernels di sistemi operativi, firmwares, drivers e simili. Rust è un linguaggio che mette in primo piano il problema dell'accesso alla memoria insieme alla velocità di esecuzione paragonabile a C/C++ e definisce delle regole molto restrittive per garantire l'assenza di tutti gli errori di accesso alla memoria (buffer overflow, race condition, invalid page fault, null pointer dereference, use after free, double free). Il linguaggio definisce un insieme di regole che adottano un approccio di tipo **fail-safe**, dove grazie al suo meccanismo dei tipi esclusivo, se non riesce a garantire la safety della memoria (assenza di tutti gli errori di accesso) mediante l'utilizzo di solo analisi statica del codice, rifiuta il programma. Per evitare ripetizioni, da ora in poi dove non specificato per i vari aspetti del linguaggio facciamo riferimento a [25].
Il compilatore di Rust è scritto in Rust stesso tramite un processo chiamato bootstrapping, che consiste nel compilare il compilatore scritto nel linguaggio Rust con un compilatore per il linguaggio Rust scritto in un altro linguaggio di programmazione (In questo caso OCaml).

Avvio e terminazione di un programma Rust

Un programma Rust inizia la sua esecuzione tramite l'invocazione della funzione main dell'eseguibile e finisce non appena questa funzione termina (anche se sono attivi altri threads):

```
1  fn main() {
2
3      // ...
4
5  } // Il programma viene terminato
```

Regole di allocazione

Le variabili (o in generale i dati) di un programma Rust possono essere allocate in tre zone di memoria differenti:

- Sullo stack (tramite la keyword **let** all'interno di una funzione).
- Nello scope globale (tramite la keyword **static**), dove vengono allocate all'inizio del programma.
- Nello heap, tramite chiamate esplicite al memory allocator o tramite strutture apposite chiamate **smart pointers** che approfondiremo nel capitolo 3.

Il rilascio della memoria su Rust non avviene tramite un garbage collector. Il linguaggio impone il rilascio della memoria posseduta da una variabile non appena il suo scope di definizione termina (out of scope). Questa regola vale sia per valori allocati esclusivamente nello stack, sia nello heap e prende il nome di RAII [13, 14].

RAII (Resource Acquisition Is Initialization):

Le risorse vengono acquisite durante l'inizializzazione di un oggetto e quando termina il suo scope di validità, il metodo distruttore dell'oggetto viene invocato automaticamente e tutte le risorse possedute vengono rilasciate.

Questa regola evita al programmatore di gestire esplicitamente il rilascio della memoria e consente di evitare la creazione di leak di memoria dovuti a un mancato rilascio. Ad esempio, se dichiariamo delle variabili all'interno di una funzione nel modo seguente:

```
1  fn main() {
2      // Per il momento ci limitiamo a dire che le stringhe sono
3      // allocate sia sullo stack, sia nello heap per spiegare
4      // il RAII. Approfondiremo la strategia completa di allocazione
5      // completa per questo tipo di dato successivamente.
6
7      // Allocazione su (stack + heap)
8      let s = String::from("Hello");
9      {
10         // Allocazione sullo stack
11         let n = 10;
12         // ...
```

```
13      } // Rilascio della memoria di n (stack)
14      // ...
15  } // Rilascio della memoria di s (stack + heap)
```

ogni volta che lo scope nel quale la variabile è definita termina, la sua memoria viene rilasciata. Nel caso di tipi composti (come ad esempio le strutture che vedremo tra poco) questa regola si applica a sua volta su tutti i campi, cioè il distruttore viene invocato automaticamente su ogni campo per rilasciare le risorse possedute a sua volta. Questo rilascio 'ricorsivo' su ogni campo è necessario, perché se il metodo distruttore venisse invocato solo sulla struttura iniziale, le risorse eventualmente associate a campi più interni non verrebbero rilasciate.
Se necessario, Rust supporta anche il rilascio anticipato della memoria associata a una variabile tramite la funzione *drop*:

```
1  fn main() {
2      let s = String::from("Hello");
3      drop(s); // Rilascio della memoria della variabile s
4  }
```

Abbiamo visto negli esempi precedenti che le variabili possono essere dichiarate con il costrutto **let**, in realtà questo vale solo se vogliamo delle variabili immutabili per cui non è possibile alterarne il contenuto. Se vogliamo dichiarare una variabile in modo da poter alterarne il contenuto, dobbiamo utilizzare il costrutto **let mut** nel modo seguente:

```
1  fn main() {
2      let n_immut = 0; // Immutabile
3      n_immut = 1; // Errore! Immutabile
4
5      let mut n_mut = 0; // Mutabile
6      n_mut = 1; // Ok
7  }
```

1.1 Il meccanismo di ownership e borrowing

Il meccanismo di ownership

Rust definisce il concetto di ownership che impone il vincolo che ogni valore abbia solo un owner (possessore) e impone lo spostamento dell'ownership (il possesso) del valore ad ogni operazione di assegnamento, disabilitando la possibilità di accedervi attraverso il vecchio owner. Ad esempio, se proviamo a utilizzare una variabile il cui valore è stato spostato per via di un assegnamento nel modo seguente:

```
1  fn main() {
2      let s1 = String::from("Hello"); // Creazione di una stringa.
3
4      // Un'assegnamento dove s1 compare alla destra ha come significato
5      // quello di spostare l'ownership sulla stringa da s1 a s2.
6      let s2 = s1;
7
8      // Scrivere semplicemente il nome di una variabile seguito da
9      // ';' equivale a leggerne il contenuto e scartare il valore
10     s1; // Errore dovuto a utilizzo
11 }
```

otteniamo un errore di questo tipo:

```
error[E0382]: use of moved value: `s1`
  --> src\main.rs:10:5
   |
2  |     let s1 = String::from("Hello"); // Creazione di una stringa.
   |         -- move occurs because `s1` has type `String`, which does not
    implement the `Copy` trait
...
6  |     let s2 = s1;
   |              -- value moved here
...
10 |     s1; // Errore dovuto a utilizzo
   |     ^^ value used here after move
```

Questo succede perché Rust adotta una semantica per spostamento [31, 28], al contrario di altri linguaggi come C++ che adottano una semantica per copia, ciò implica che l'assegnamento non porta

a una copia o condivisione del valore tra più variabili, ma sposta i permessi di accesso al valore su di un'altra variabile. Nel nostro esempio, stiamo provando a utilizzare la variabile *s1* dopo che la ownership sulla stringa è stata trasferita a *s2* e *s1* ha perso ogni diritto di accesso.
Questa cosa è vera in generale per tipi composti e definiti dall'utente, ma non vale per alcuni tipi come ad esempio gli scalari. I tipi scalari (cioè gli interi, i numeri in virgola mobile, caratteri e booleani) vengono passati direttamente per copia e quindi negli assegnamenti non viene spostata la ownership del valore. Infatti se proviamo a assegnare a una variabile *n2*, una variabile *n1* che contiene un numero nel modo seguente:

```
1  fn main() {
2      let n1 = 1;
3      let n2 = n1;
4
5      n1; // Ok
6  }
```

se proviamo a utilizzare la variabile *n1* dopo l'assegnamento, non viene generato nessun errore, perché alla variabile *n2* viene assegnata una copia del valore di *n1*. L'esempio precedente a quest'ultimo, fornisce anche un messaggio che informa che 'il tipo String non implementa il trait Copy'. Vedremo i traits quando parleremo dei tipi, quindi per ora ci limitiamo spiegare questo messaggio di errore dicendo che il tipo non definisce un meccanismo per creare una copia del valore da assegnare, e quindi il valore della variabile può essere solamente trasferito alla nuova variabile tramite ownership (come succede nell'esempio).

Il meccanismo di borrowing

Il passaggio della ownership avviene anche quando si opera con le funzioni. Infatti quando passiamo un valore come parametro che non può essere passato per copia come ad esempio una stringa, in automatico l'ownership sul valore viene trasferita alla funzione e non è più possibile accedere al valore dopo l'invocazione tramite la vecchia variabile. Ad esempio, se passiamo una variabile che contiene una stringa a una funzione e proviamo a usarla dopo l'invocazione nel modo seguente:

```
1  // La funzione prende in ingresso una stringa
2  // tramite trasferimento di ownership.
```

```
3  fn print_it(s: String) {
4      // Invocazione di una macro per la stampa a schermo
5      println!("{}", s) // Stampa la stringa s
6  }
7
8  fn main() {
9      let s1 = String::from("Hello");
10     print_it(s1);
11     // Errore! La variabile s1 non ha più
12     // i diritti di accesso al valore
13     s1;
14 }
```

otteniamo un errore simile al precedente. Questo accade perché *s1* non è più l'owner della stringa, infatti l'ownership è stata trasferita al parametro *s* della funzione *print_it*, che a sua volta ha rilasciato la memoria associata alla stringa come imposto dal meccanismo di rilascio della memoria, al termine del corpo della funzione (*print_it*).
Lo stesso meccanismo è sfruttato dalla funzione *drop* vista poco fa per rilasciare la memoria in anticipo, infatti la sua implementazione è la seguente [9]:

```
1  // Si può operare con tipi generici dichiarando
2  // i generics tra parentesi angolate "<>".
3  fn drop<T>(v: T) { }
```

Il metodo si limita a prendere l'ownership sul valore di tipo generico T passato come parametro e a rilasciare immediatamente la memoria come imposto dal RAII [13, 14] definendo un corpo vuoto. Da questa dichiarazione di funzione possiamo anche notare che il linguaggio supporta i generics per operazione con la stessa funzione su tipi diversi.

Il vincolo di avere un solo owner per un valore è molto restrittivo e non permetterebbe di riutilizzare il valore per più funzioni, quindi Rust insieme al meccanismo di ownership offre anche un meccanismo di borrowing (prestiti). Un borrow su di una variabile o in generale su un valore si effettua con il costrutto "**&**" o "**&mut**" e permette di dare un accesso temporaneo al valore della

variabile rispettivamente con solo permessi di sola lettura ("**&**") oppure con permessi di lettura e scrittura ("**&mut**"). Ad esempio, è possibile dare accesso temporaneo a una stringa contenuta in una variabile nel modo seguente:

```
1  // La funzione prende in ingresso un riferimento a una stringa
2  fn print_it(s: &String) {
3      // Invocazione di una macro per la stampa a schermo.
4      println!("S={}", s); // Stampa la stringa s
5  }
6
7  fn main() {
8      let s1 = String::from("Hello"); // Creazione di una stringa.
9      print_it(&s1); // Borrow. Stampa: S=Hello.
10     s1; // Ok! La stringa è sempre valida.
11 }
```

In questo modo è possibile utilizzare la stringa *s1* anche dopo l'invocazione di funzione, perché l'accesso al valore è stato 'prestato' tramite riferimento alla funzione solo per la durata del suo corpo. Notiamo che per permettere il passaggio di un parametro tramite il meccanismo di borrowing, è necessario modificare la firma della funzione facendo precedere il tipo del parametro dal costrutto "**&**" o "**&mut**" per accettare rispettivamente riferimenti in sola lettura oppure in lettura e scrittura creati tramite il meccanismo di borrowing. Precisiamo che il meccanismo per concedere accesso temporaneo a un valore si chiama borrowing, ma a tutti gli effetti quello che stiamo creando e passiamo alla funzione è un riferimento al valore, quindi da ora in poi quando parleremo di riferimenti ci riferiremo implicitamente sempre al meccanismo di borrowing per la loro creazione [6].

I riferimenti creati tramite il meccanismo di borrowing tramite i costrutti "**&**" e "**&mut**" seguono regole simili a **let** e **let mut**, solo che la loro creazione a differenza delle variabili è soggetta a restrizioni come vincolato dalla regola AXM [33]:

Regola AXM (Aliasing Xor Mutability):

La regola AXM impone, che se un valore è soggetto a condivisione (alias) non può essere

accessibile in modo mutabile (mutability) allo stesso tempo e se un valore è accessibile in modo mutabile non può essere condiviso.

Ad esempio, se una variabile è dichiarata come immutabile tramite il costrutto **let**, possiamo creare un numero arbitrario di riferimenti immutabili (alias) tramite il costrutto "**&**":

```
1  fn main() {
2      let n_immut = 0; // immutabile.
3      let r1_immut = &n_immut;
4      // ...
5      let rn_immute = &n_immut;
6
7      // Stampa: R1=0, RN=0
8      println!("R1={}, RN={}", r1_immut, rn_immut);
9  }
```

Se invece la variabile è dichiarata come mutabile, è possibile creare un numero arbitrario di riferimenti immutabili (alias) o un solo riferimento mutabile (mutability) per volta. Questa cosa consente di modificare il valore contenuto anche se non abbiamo l'ownership diretta sul valore e garantisce di essere gli unici ad avere accesso su quel valore. Per creare un riferimento mutabile a una variabile mutabile si utilizza il costrutto "**&mut**":

```
1  fn main() {
2      let mut n_mut = 0; // Mutabile
3      let r_mut = &mut n_mut;
4
5      *r_mut = 1; // Assegnamento tramite dereferenziazione
6
7      println!("R={}", r_mut); // Stampa: R=1
8  }
```

Inoltre, nel caso di sovrascrittura del valore tramite riferimento mutabile, la memoria associata al vecchio valore viene immediatamente rilasciata al momento dell'assegnamento:

```
1  fn main() {
2      let mut s_mut = String::from("A");
```

```
3
4       // Rilascio della memoria della stringa "A"
5       // e sostituzione del valore con la stringa "B".
6       *r_mut = String::from("B");
7
8       println!("R={}", r_mut); // Stampa: R=B
9   }
```

Non è possibile invece creare altri riferimenti mutabili mentre un altro riferimento mutabile è in utilizzo e non è possibile creare riferimenti con permessi di accesso diversi, mentre un riferimento di un tipo è ancora utilizzato nel codice successivamente al momento della creazione (come imposto dalla regola AXM):

```
1   fn main() {
2       let mut n_mut = 0; // Mutabile.
3       let r1_mut = &mut n_mut;
4
5       // Errore! Il riferimento r1_mut (mutabile) ed è ancora in uso.
6       let r2_mut = &mut n_mut;
7
8       // Errore! Non è possibile creare riferimenti immutabili mentre un
9       // riferimento mutabile è ancora in uso. Questa cosa vale anche
10      // per il viceversa, cioè non è possibile creare riferimenti
11      // mutabili mentre sono presenti riferimenti immutabili.
12      let r1_immut = &n_mut;
13
14      println!("R={}", r1_mut); // Utilizzo di r1_mut (mutabile);
15  }
```

Inoltre, quando su una variabile è stato effettuato un borrow e un riferimento è ancora in utilizzo, ci sono delle restrizioni anche sull'accesso diretto alla variabile a seconda del tipo di riferimento attivo. Ad esempio, se è presente un riferimento mutabile, non si può accedere alla variabile:

```rust
fn main() {
    let mut n_mut = 0; // Mutabile.
    let r_mut = &mut n_mut; // riferimento mutabile.

    // Errore! Un riferimento mutabile (r_mut) è ancora in uso.
    println!("N={}", n_mut);
    println!("R={}", r_mut); // Utilizzo riferimento mutabile.
}
```

Questa cosa è corretta altrimenti sarebbe possibile violare la regola AXM e modificare o leggere il valore mentre è attivo un altro accesso mutabile. La stessa regola si applica anche in presenza di riferimenti immutabili. Ad esempio, se è presente un riferimento immutabile, anche se la variabile è dichiarata come mutabile, è possibile accedere alla variabile solo in lettura e non è possibile modificare il valore:

```rust
fn main() {
    let mut n_mut = 0; // Mutabile.
    let r_immut = &n_mut;

    println("N={}", r_immut); // Ok! Solo lettura.

    // Errore! Un riferimento immutabile è ancora in uso.
    n_mut = 1;

    println!("R={}", r_immut); // Utilizzo riferimento immutabile.
}
```

Inoltre, non è possibile muovere l'ownership di un valore attraverso un riferimento, perché attraverso un riferimento viene garantito un accesso temporaneo e non è possibile forzare un accesso permanente. Questa regola si limita ai tipi che non sono copiabili, per i tipi copiabili come gli scalari non viene spostata l'ownership e vengono copiati direttamente anche in caso di accesso tramite riferimento. Ad esempio, si può assegnare un riferimento a un valore scalare, ma non si può assegnare un riferimento a un valore di tipo stringa:

```rust
fn main() {
    let mut n_mut = 0; // Mutabile scalare.
    let mut s_mut = String::new(); // Mutabile stringa.

    let rn_immut = &n_mut;
    let rs_mut = &mut s_mut;

    let other_n = *rn_immut; // Ok! Passato tramite copia.

    // Errore! Non si può spostare l'ownership tramite
    // riferimento e il tipo String non è copiabile.
    let other_s = *rs_mut;
}
```

Controllo del flusso

Il meccanismo di ownership e borrowing ha effetto anche nei costrutti di flusso come ad esempio **for** e **if**, ma non si limita a questi e si applica in generale a tutti in modo analogo. Nel costrutto **for** è possibile creare riferimenti all'interno del corpo, ma tutti i riferimenti creati vengono distrutti al termine di ogni iterazione. Inoltre, non è possibile spostare l'ownership di un valore senza ripristinarlo prima dell'iterazione successiva. Per chiarire questo concetto si osservi l'esempio seguente:

```rust
fn main() {
    let mut v = String::from("Hello");
    for i in 0..10 {
        // Ok si può fare, ma
        let k = v;

        // solo se si ripristina l'ownership su un
        // valore prima della prossima iterazione.
        v = String::from("Hello");

```

```
11          let r = &v;
12          println!("R={}", r); // Stampa 10 volte: R=Hello
13      } // Memoria di k e r rilasciata.
14  } // Memoria di v rilasciata.
```

Con Il costrutto **if** abbiamo il vincolo di rispettare il meccanismo di ownership e borrowing in tutti i possibili cammini di esecuzione. Ad esempio non è corretto muovere l'ownership di un valore da una variabile a un'altra in un ramo e utilizzare la variabile che ha ceduto l'ownership successivamente:

```
1  fn main() {
2      // Prendi il numero di argomenti passati al programma.
3      let n = args().count();
4      let mut v = String::from("Hello");
5
6      if n > 10 {
7          let k = v; // Spostamento di ownership.
8          println!("K: {}", k);
9      } else {
10         let s = &v; // Creazione di un riferimento.
11         println!("S: {}", s);
12     }
13
14     // Errore! Un ramo dell'if muove l'ownership sul valore
15     // associato alla variabile v, quindi a questo punto del
16     // codice potrebbe non avere più i permessi per accedere.
17     println!("{}", v);
18 }
```

1.2 I tipi principali

Gli scalari

In Rust sono presenti i tipi scalari e hanno una dimensione fissa indipendentemente dall'architet-

tura, infatti il linguaggio permette di definire la dimensione della rappresentazione e se vogliamo operare con o senza segno.
I tipi scalari principali sono:

- Gli interi con e senza segno a partire da 1 byte fino a 16 bytes, che sono indicati tramite il numero di bits della rappresentazione (*i8, ... i128, u8, ... u128*):

```
1  let i1: i8 = -1;
2  let i2: u128 = 1;
```

- I numeri in virgola mobile sono presenti su 4 e 8 bytes, che sono indicati anche in questo caso dal numero di bits della rappresentazione (*f32, f64*):

```
1  let f1: f32 = -1.0;
2  let f2: f64 = 1.0;
```

- I tipi bool e char per rappresentare i booleani e i caratteri, che sono rappresentati rispettivamente su 1 byte (bool) e 4 bytes (char):

```
1  let b1: bool = true
2  let c1: char = 'a'
```

- Il tipo usize, è l'unico tipo che cambia di dimensione a seconda dell'architettura, infatti ha una dimensione di 4 bytes su architetture a 32 bits e 8 bytes su architetture a 64 bits. Può rappresentare numeri senza segno e solitamente è utilizzato per salvare indirizzi di memoria generici che non hanno un tipo ben definito.

```
1  let addr: usize = 0xf;
```

Le stringhe

Le stringhe in Rust sono un tipo speciale, infatti si dividono in tre categorie:

- Gli string literals, che corrispondono alle stringhe statiche, cioè le stringhe il cui contenuto e lunghezza sono conosciuti a tempo di compilazione. Queste stringhe non sono modificabili, vengono scritte direttamente dentro l'eseguibile al momento della compilazione, è possibile accederci da programma solo tramite un riferimento immutabile e hanno validità per tutta la durata del programma.

```rust
let literal: &str = "Hello";
println!("{}", literal); // Stampa: Hello.
```

- il tipo String, che rappresenta una stringa dinamica, il cui contenuto è allocato nello heap e può cambiare a tempo di esecuzione. A differenza di altri linguaggi di programmazione come Java dove anche le stringhe create dinamicamente sono immutabili, in Rust le operazioni di modifica su stringhe non creano nuove istanze, bensì modificano la stessa istanza.

```rust
let mut s = String::new(); // Si creano con String::new().
s.push_str("Hello"); // Aggiunta in coda alla stringa "Hello".
println!("{}", s); // Stampa: Hello.
```

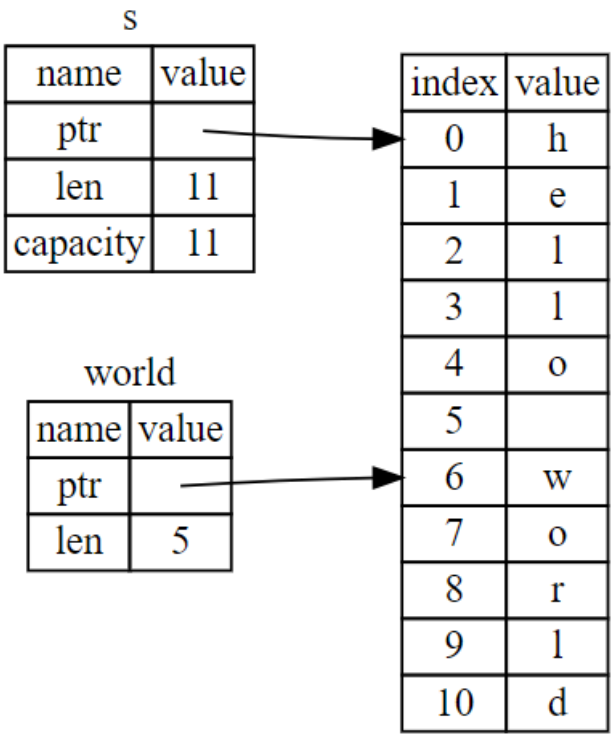
- Il tipo string slice, che rappresenta una parte di una stringa che può essere acceduta in modo mutabile o immutabile a seconda dei permessi di accesso.

```rust
let mut s = String::new();
s.push_str("Hello");
let slice = &s[0..3]; // Sintassi per creare uno slice.
println!("'{}'", slice); // Stampa: Hel.
```

La tecnica di allocazione della memoria per gestire le stringhe dinamiche è progettata in modo da garantire la safety della memoria. Nel dettaglio la tecnica prevede di allocare sullo stack informazioni aggiuntive come l'indirizzo di memoria di inizio, la lunghezza e la capacità della stringa o dello slice, per poter controllare i limiti consentiti quando viene acceduta e di allocare il buffer per la stringa nello heap. Ad esempio, la seguente stringa e il relativo slice:

```rust
let mut s = String::new("Hello world");
let world = &s[6..11];
```

Sono allocati, rispettivamente sullo stack e sullo heap nel modo seguente:



Nel caso di string literal invece di avere il buffer allocato nell'heap, esse sono scritte direttamente dentro l'eseguibile e sullo stack viene memorizzato l'indirizzo di memoria, la lunghezza e la capacità.

Le tuple e gli array

Nel linguaggio sono presenti anche i tipi tupla e tipi array per rappresentare rispettivamente una collezione di dati di tipo eterogeneo o omogeneo di lunghezza fissa e conosciuta a tempo di compilazione:

```
1  let tuple = (1, 'a', "Hello");
2  let array = [1, 2, 3];
```

Non è possibile definire un array in funzione di una variabile il cui valore non è costante:

```
1  // Ok. Creazione di un array di tipo i32 di
2  // lunghezza 10 con tutti valori inizializzati a 0.
3  let array = [0; 10];
4
5  // Prendi il numero di argomenti passati al programma.
6  let n = args().count();
```

```
7  // Errore! La variabile  n non assume un
8  // valore costante a tempo di compilazione.
9  let array = [0; n];
```

Il motivo per cui questa dichiarazione non è possibile, deriva dal problema che il compilatore utilizza lo stack per allocare questi tipi e non sa quanto spazio riservare, dato che può cambiare. Inoltre per garantire la safety della memoria, con gli accessi ad array e tuple, Rust introduce dei controlli a tempo di esecuzione per controllare che l'indice acceduto sia effettivamente valido, facendo terminare immediatamente il thread nel caso in cui l'indice acceduto è negativo o supera la lunghezza effettiva:

```
1  let array = [0, 1, 2];
2  // Ok, L'indice cade dentro l'array.
3  println!("V1: {}", array[0]);
4  // Errore a tempo di esecuzione la lunghezza dell'array è 3,
5  // ma l'indice acceduto è 10. Il thread viene terminato con un errore
6  // oppure il programma viene terminato nel caso di main thread.
7  println!("V10: {}", array[10]);
```

Le enumerazioni, le strutture e i traits

Il linguaggio definisce il concetto enumerazione, che sono un modo per restringere il numero di valori che possono essere assunti da una variabile, da un campo di una struttura, da un parametro o dal valore di ritorno di una funzione. Le enumerazioni si dichiarano con la keyword **enum**:

```
1  enum IPVersion {
2      V4,
3      V6
4  }
```

Le enumerazioni possono anche contenere dei valori ed è possibile operare con tipi diversi utilizzando i generics. Ad esempio, il linguaggio per gestire la presenza e l'assenza di valore utilizza la enumerazione **Option** che è definita nel modo seguente:

```
1  enum Option<T> {
2      None,
```

```
3       Some(T)
4   }
```

Per accedere a un valore di una enumerazione è necessario fare pattern matching:

```
1   fn main() {
2       let mut opt = None; // Inizializziamo con None.
3       opt = Some(3); // Insieriamo Some con valore 3.
4
5       match opt {
6           None => {
7               // Non è presente un valore.
8               println!("Val=None");
9           },
10          Some(val) => {
11              // Ok è presente un valore,
12              // si può accedere tramite 'val'.
13              println!("Val={}", val);
14          }
15      }
16
17      // Stampa: Val=3.
18  }
```

In questo esempio, nel caso di Some è presente un valore ed è possibile accedervi tramite la variabile *val*, mentre nel caso di None non è presente nessun valore.

Nel linguaggio sono presenti anche le strutture per raggruppare un insieme di campi di grandezza fissa e di tipo diverso. Rust riprende alcuni concetti della programmazione orientata agli oggetti e su di esse è possibile definire anche dei metodi come nel caso delle classi di altri linguaggi come ad esempio C++/Java, ma non è possibile estenderle per ereditare proprietà e metodi, perché Rust non definisce un concetto di ereditarietà tra strutture. Le strutture si dichiarano tramite la keyword **struct** e si possono definire dei metodi che operano su di esse, definendo delle funzioni all'interno di un blocco **impl**:

```
1  struct MyStruct {
2      n: i32, // Intero.
3      s: String // Stringa.
4  }
5
6  impl MyStruct {
7      // Il metodo costruttore tipicamente si chiama new.
8      fn new() -> MyStruct {
9          // Non mettere il punto e virgola dopo un'espressione equivale
10         // a eseguire il return del valore restituito dall'espressione
11         // (in questo caso l'istanza della struttura appena creata).
12         MyStruct {
13             n: 0,
14             s: String::new()
15         }
16     }
17
18     // Per modificare un campo di un'istanza è necessario richiedere
19     // un riferimento mutabile all'istanza stessa come parametro (self).
20     fn incr(&mut self) {
21         self.n += 1;
22     }
23 }
```

Non è possibile definire solo alcuni campi mutabili e altri immutabili, tutti i campi sono mutabili solo se l'istanza è accessibile tramite un accesso mutabile altrimenti sono immutabili. Nel nostro esempio, il metodo *incr* deve incrementare il valore del campo *n* all'interno dell'istanza della struttura e quindi ha bisogno che gli venga passato un riferimento mutabile all'istanza stessa (**&mut self**). Potremmo pensare che questo approccio di accesso ai campi non sia molto corretto, in realtà il compilatore vieta l'accesso ai campi e ai metodi della struttura dall'esterno se non sono dichiarati pubblici (con la keyword **pub**) e quindi se è necessario definirne solo alcuni campi mutabili, è sufficiente definire dei metodi pubblici che modificano solo quelli di interesse.

Rust supporta i traits, che sono simili alle interfacce del linguaggio Java e permettono di definire il comportamento delle strutture che li implementano. Un trait è una collezione di firme di funzioni e metodi, eventualmente completati con un'implementazione di default. Un trait si dichiara con la keyword **trait** e su un tipo (strutture, scalari e così via) con la keyword **impl** nel modo seguente (assumiamo che X e Y siano strutture):

```
1  // Definizione. Si possono implementare sulle strutture,
2  // ma è possibile implementarli su qualsiasi tipo.
3  trait Summary {
4      fn summarize(&self) -> String;
5  }
6
7  // Implementazione del trait sulla struttura X.
8  impl Summary for X {
9      fn summarize(&self) -> String {
10         String::from("X")
11     }
12 }
13
14 // Implementazione del trait sulla struttura Y.
15 impl Summary for Y {
16     fn summarize(&self) -> String {
17         String::from("Y")
18     }
19 }
20
21 // Implementazione del trait sul tipo i32 (scalare).
22 impl Summary for i32 {
23     fn summarize(&self) -> String {
24         String::from("I32")
25     }
```

```
26 }
```

I trait possono servire sia per evitare duplicazione del codice, definendo un metodo in un trait con un'implementazione di base per i tipi che implementano quel trait, oppure può essere usato per definire la firma dei metodi che una struttura deve implementare. Quest'ultimo aspetto consente di utilizzare un tipo dove sono richiesti tipi che possiedono determinati metodi in comune. Ad esempio, se vogliamo definire una funzione che accetta in ingresso tutti i tipi che implementano il trait Summary, possiamo imporre che il tipo passato come parametro lo implementi nel modo seguente:

```
1  // Imponi che il tipo in ingresso implementi (impl) il trait Summary.
2  fn print_summary(s: &impl Summary) {
3      println!("{}", s.summarize())
4  }
```

Inoltre, i trait possono essere implementati anche in modo condizionale in base al tipo con cui viene istanziato un generic (assumiamo che le strutture X e Y definiscano un generic T nella loro definizione):

```
1  // Implementa Summary solo se T è istanziato con un tipo che implementa
2  // il trait PartialOrd, cioè se il tipo supporta l'ordinamento parziale
3  impl<T: PartialOrd> Summary for X<T> {
4      fn summarize(&self) -> String {
5          String::new("X")
6      }
7  }
8
9  // Implementa solo se T è istanziato a i32
10 impl Summary for Y<i32> {
11     fn summarize(&self) -> String {
12         String::new("Y")
13     }
14 }
```

Con il costrutto per generics **T:** è possibile imporre che il tipo a cui viene istanziato il generic implementi uno o più trait a sua volta affinché i metodi definiti nel blocco impl si possano usare (Nel nostro esempio PartialOrd). Se invece il generic viene istanziato con un tipo concreto, il trait viene implementato solo se il generic viene istanziato esattamente a quel determinato tipo (nel nostro esempio i32).

Nel linguaggio sono presenti alcuni trait speciali per effettuare l'overloading degli operatori e sono presenti anche altri traits speciali come Clone e Copy [8] che permettono rispettivamente di creare una deep copy di un valore (Clone) tramite un metodo *clone* o di creare una shallow copy del valore (Copy) durante un assegnamento per implementare il passaggio per copia:

```
1  struct X {
2      v: i32
3  }
4
5  impl Clone for X {
6      // Il trait Clone permette di creare una deep copy invocando il
7      // metodo clone e permette di copiare anche alterando i valori.
8      fn clone(&self) -> Self {
9          // La copia avrà valore aumentato di 1.
10         X {
11             v: (self.v + 1)
12         }
13     }
14 }
15
16 // Il trait Copy permette di creare una shallow copy durante un
17 // assegnamento. Non si possono alterare i valori dei campi della
18 // copia, quindi la copia avrà esattamente gli stessi valori nei campi.
19 impl Copy for X { }
20
21 fn main() {
```

```
22      let x1 = X { v: 1 };
23      // Shallow copy (copia bit a bit dei campi della struttura).
24      let x2 = x1;
25      // Deep copy (copia invocando il metodo clone).
26      let x3 = x1.clone();
27
28      // Shallow copy print.
29      println!("X1={}, X2={}", x1.v, x2.v); // Stampa: X1=1, X2=1.
30      // Deep copy print.
31      println!("X1={}, X3={}", x1.v, x3.v); // Stampa: X1=1, X3=2.
32  }
```

Una struttura (in generale un tipo) può implementare il trait Clone sempre, mentre può implementare il trait Copy se e solo se tutti i campi della struttura implementano Copy e non è possibile forzarne l'implementazione.
Il motivo per cui prima abbiamo detto che gli scalari vengono passati per copia e non per ownership è perché tutti gli scalari implementano il trait Copy.

Le funzioni e le chiusure

Le funzioni e le chiusure sono anche esse dei tipi e possono essere create a livello globale o all'interno di altre funzioni e possono essere passate come parametri a altre funzioni o restituite come risultati. Le funzioni si dichiarano tramite la keyword **fn**, mentre le chiusure con il costrutto "||" nel modo seguente:

```
1   // Definizione di una funzione con parametro x di tipo i32
2   fn outer(x: i32) {
3       // Funzione dentro una funzione che ritorna un intero.
4       fn inner() -> i32 {
5           1 // Ritorna 1 (non è presente il ";").
6       }
7
8       // Invoca la funzione inner e salva
9       // il risultato nella variabile 'k'.
```

```
10      let k = inner();
11  }
12
13  // Funzione che prende in ingresso una funzione
14  // che chiede in ingresso un valore di tipo i32.
15  fn as_param(f: fn(i32) -> ()) {
16      f(1); // Invocazione della funzione passata come parametro.
17  }
18
19  // Funzione che restituisce una funzione.
20  fn as_return() -> (fn(i32) -> ()) {
21      outer // Ritorna la funzione 'outer' come risultato.
22  }
23
24  // Definizione di una chiusura con parametro x1 di tipo i32.
25  let closure = |x: i32| -> i32 {
26      x + 1 // Ritorna il successore.
27  }
```

Le chiusure nel linguaggio sono diverse dalle funzioni, infatti le funzioni non possono catturare l'ambiente esterno e possono accedere solo a variabili e funzioni dichiarate nello scope globale, mentre le chiusure possono catturare l'ambiente esterno e a seconda di come interagiscono con esso implementano automaticamente determinati traits [15].

Le chiusure che utilizzano variabili catturate nell'ambiente esterno al corpo della chiusura alla destra di un assegnamento (cioè spostano l'ownership), spostano l'ownership dei valori catturati alla chiusura stessa e implementano solo il trait FnOnce. Questo implica che possono essere invocate solo una volta e si possono passare come parametro a altre funzioni che richiedono chiusure di FnOnce. Tutte le chiusure implementano automaticamente almeno questo trait:

```
1  fn main() {
2      let mut s = String::from("A");
3      // Il valore della variabile s è catturato tramite
```

```
4       // ownership, perché compare alla destra di un
5       // assegnamento all'interno della chiusura.
6       let closure = || {
7           // Il valore catturato rimane quello assegnato al
8           // momento della dichiarazione della chiusura.
9           let new_owner = s;
10          println!("{}", new_owner); // Stampa A.
11      };
12
13      // Errore! La chiusura è diventata l'owner della stringa.
14      println!("{}", s);
15
16      // Se sostituiamo il comando precedente con il
17      // seguente il programma compila correttamente
18      s = String::from("B");
19
20      // Ok! La variabile s ha l'ownership su una nuova stringa.
21      println!("{}", s); // Stampa B
22
23      closure(); // Invocazione della chiusura.
24  }
```

Le chiusure che accedono in lettura e scrittura, catturano l'ambiente tramite borrow mutabile e implementano i trait FnOnce e FnMut. Questo implica che possono essere invocate un numero infinito di volte e si possono passare come parametro a altre funzioni che richiedono chiusure di tipo FnOnce o FnMut:

```
1   fn main() {
2       let mut s = String::from("A");
3       // Accede a s in scrittura.
4       let closure = || {
5           s = String::from("B");
```

```
6           println!("{}", s); // Stampa B.
7       }
8
9       // Errore! la variabile s è catturata tramite borrow mutabile
10      // dalla chiusura, quindi non si può accedere al valore
11      // visto che essa viene invocata successivamente.
12      println!("{}", s);
13
14      closure(); // Invocazione della chiusura.
15  }
```

Le chiusure che accedono solo in lettura alle variabili catturate nell'ambiente esterno al corpo della chiusura, catturano le variabili tramite borrow immutabile e implementano i trait FnOnce, Fn e FnMut. Questo implica che possono essere invocate un numero infinito di volte e possono essere passate come parametro a altre funzioni che richiedono chiusure di tipo FnOnce, Fn o FnMut:

```
1   fn main() {
2       let mut s = String::new();
3       // Accede a s in sola lettura.
4       let c = || {
5           println!("{}", s);
6       }
7
8       println!("{}", s); // Ok
9
10      // Errore! La variabile s è catturata tramite borrow immutabile
11      // dalla chiusura, quindi è vietato modificare il valore
12      // dato che la chiusura viene invocata successivamente.
13      s = String::new();
14
15      closure(); // Invocazione della chiusura.
16  }
```

Precisiamo che se ci sono dei borrow incompatibili (mutabili e immutabili) la chiusura implementa FnOnce e FnMut. Nel caso in cui esiste anche solo una occorrenza nel codice della chiusura dove viene spostata l'ownership, automaticamente la chiusura implementa solo FnOnce. Questo non implica che se una variabile viene catturata secondo un modo (ownership o borrow) automaticamente anche tutte le altre saranno catturate allo stesso modo, infatti ogni variabile viene catturata mediante i permessi minimi necessari alla chiusura:

```
1  fn main() {
2      let mut s1 = String::from("A");
3      let mut s2 = String::from("B");
4
5      let closure = || {
6          // Cattura tramite ownership.
7          let new_owner = s1;
8          // Cattura tramite borrow immutabile.
9          println!("{}", s2);
10     }
11
12     // Errore! l'ownership sulla stringa è stato spostato.
13     println!("{}", s1);
14
15     // Ok! la variabile s2 è catturata tramite borrow immutabile.
16     println!("{}", s2);
17
18     closure();
19 }
```

1.3 Multi-threading

Rust supporta il multi-threading, ma in modo differente rispetto a altri linguaggi come ad esempio C++/Java. Per garantire la safety della memoria anche in questo ambito, Rust impone delle regole

sui valori che possono essere inviati a altri threads o condivisi tra più threads contemporaneamente.

I threads

I threads possono essere creati tramite la funzione *thread::spawn* [11] e richiedono una chiusura di tipo FnOnce (invocabile almeno una volta) come parametro, inoltre per passare dei parametri a un thread è sufficiente che la chiusura catturi le variabili di interesse come ambiente esterno:

```
1  fn main() {
2      let mut x = 0;
3      // la keyword move forza lo spostamento di ownership
4      // invece di lasciare la cattura tramite borrow mutabile
5      thread::spawn(move || {
6          x += 1;
7          println!("{}", x); // Stampa 1
8      })
9
10     println!("Hello"); // Stampa Hello
11 }
```

Come notiamo, in questo caso è stata utilizzata la keyword **move** prima della dichiarazione della chiusura, questa keyword forza la cattura dell'ambiente esterno tramite spostamento di ownership alla chiusura. Questa keyword è necessaria, perché la variabile catturata risiede sullo stack di un altro thread e quindi la sua memoria potrebbe essere rilasciata prima del termine dell'esecuzione dell'altro thread. Questo esempio stampa come risultato '1' e 'Hello' in ordine casuale a seconda dell'ordine di schedulazione dei thread e a seconda di chi riesce ad acquisire la lock sulla console. Inoltre l'esempio 'invia' uno scalare a un altro thread e questo non causa nessun problema, ma non tutti i tipi in Rust possono essere utilizzati in ambito multi-thread. Infatti per essere utilizzati in ambito multi-thread, Rust richiede che i tipi catturati dalla chiusura implementino alcuni trait speciali.

Il trait Send

Il trait Send è un trait speciale di tipo 'marcatore' perché non definisce nessun metodo da implementare e serve a indicare che un tipo di dato può essere inviato a un altro thread tramite trasferimento

di ownership. I tipi scalari e le stringhe lo implementano automaticamente in tutti i contesti di utilizzo:

```
1  fn main() {
2      let mut x = 0;
3      let mut s = String::new();
4      thread::spawn(move || {
5          x += 1; // Somma 1
6          s.push('!'); // Aggiungi in fondo alla stringa il carattere '!'.
7          println!("{}{}", x, s); //Stampa 1!
8      });
9
10     println!("Hello"); // Stampa Hello
11 }
```

Trasferire degli scalari o delle stringhe dallo stack di un thread allo stack di un altro thread è perfettamente corretto e non porta a nessun possibile errore di accesso alla memoria.

Altri tipi lo implementano in modo condizionale:

```
1  impl<T: Send, A: Send> Send for Vec<T, A>
```

Il tipo Vec per manipolare sequenza dinamiche di dati, implementa il trait Send se e solo se il tipo contenuto (T) lo implementa a sua volta. L'altro generic (A) non riguarda gli elementi nel vettore e non è di nostro interesse, perché riguarda il memory allocator associato all'istanza del vettore. Potremmo pensare che non ci sono tipi che non possono essere inviati a un altro thread, perché alla fine stiamo solamente trasferendo l'ownership da un thread a un altro, in realtà esistono alcune eccezioni come ad esempio i tipi che permettono di avere ownership multipla senza meccanismi di sincronizzazione, ma ne parleremo nel capitolo 3.

Il trait Sync

Il trait Sync è un altro trait speciale che non definisce nessun metodo da implementare e serve a indicare che un tipo di dato può essere condiviso tra threads diversi contemporaneamente tramite

riferimento immutabile. Un tipo **T** implementa il trait Sync se e solo se il suo riferimento immutabile **&T** implementa Send, cioè un tipo implementa Sync se e solo se condividere il suo riferimento immutabile con altri thread per avere accesso in lettura concorrente non porta alla violazione della regola AXM. Anche questo trait è implementato da molti tipi. Ad esempio, gli scalari e le stringhe lo implementano in ogni contesto di utilizzo:

```
1  static s: String = String::new(); // Stringa vuota.
2  fn main() {
3      for i in 0..5 {
4          let r = &s;
5          // Cattura tramite ownership la variabile che
6          // contiene il riferimento immutabile a 's'.
7          thread::spawn(move || {
8              println!("R={}", r); // Stampa 5 volte: R=.
9          });
10     }
11 }
```

In questo caso creiamo un riferimento immutabile alla stringa statica e chiediamo alla chiusura di catturare la variabile r contenente il riferimento alla stringa tramite ownership. Il motivo per cui la stringa è stata dichiarata nello scope globale non è per una preferenza, ma è obbligatorio in questo caso. Se infatti proviamo a passare una stringa dichiarata all'interno della funzione:

```
1  fn main() {
2      let s = String::new(); // Stringa vuota.
3      for i in 0..5 {
4          let r = &s;
5          thread::spawn(move || {
6              println!("R={}", r);
7          });
8      }
9  }
```

otteniamo un errore di questo tipo:

```
error[E0597]: `s` does not live long enough
  --> src\main.rs:6:17
   |
6  |           let r = &s;
   |                   ^^ borrowed value does not live long enough
7  | /         thread::spawn(move || {
8  | |             println!("R={}", r);
9  | |         });
   | |__________- argument requires that `s` is borrowed for `'static`
10 |       }
11 |   }
   |   - `s` dropped here while still borrowed
```

Questo errore ci avvisa che la stringa è allocata sullo stack del thread (tramite let nella funzione main) e potrebbe succedere che il thread che la possiede termini la sua esecuzione e che la memoria associata alla stringa venga rilasciata prima che tutti i threads che hanno accesso ad essa tramite riferimento abbiano terminato di utilizzarla. Anche in questo caso esistono delle eccezioni di tipi che non possono implementare Sync, questo succede tipicamente quando un tipo offre **interior mutability**, cioè quando è possibile accedere in modo mutabile al valore anche attraverso un riferimento immutabile, ma approfondiremo questo concetto nel capitolo 3.

Questi traits sono speciali ed è molto importante che la loro implementazione sia effettuata solo da tipi che effettivamente sono thread safe. Questo è garantito dal fatto che non possono essere implementati normalmente usando un blocco **impl**, ma sono implementati implicitamente dal compilatore se e solo se tutti i campi all'interno di una struttura a loro volta implementano i rispettivi trait (Send e/o Sync). Tuttavia, come vedremo tra poco, è comunque possibile implementare questi trait forzatamente utilizzando codice **unsafe**.

1.4 Rust unsafe

Rust, oltre a permettere di scrivere codice safe attraverso un insieme di operazioni e regole che garantiscono l'assenza degli errori di memoria più comuni, permette di eseguire alcune operazioni aggiuntive che non garantiscono la safety della memoria. Questo tipo di operazioni non possono essere eseguite normalmente e devono essere eseguite dentro un blocco unsafe con la keyword

unsafe. Precisiamo che l'utilizzo dei blocchi unsafe, non disabilita le normali regole del linguaggio, ma introduce la possibilità di effettuare alcune operazioni aggiuntive non soggette a questi vincoli. La scrittura di codice che utilizza queste operazioni aggiuntive che non garantiscono la safety della memoria all'interno di un blocco unsafe, prende il nome di codice unsafe.

Dereferenziazione di puntatori

L'abilità principale che ci concede un blocco unsafe è quella di dereferenziare un puntatore:

```
1  fn main() {
2      // Casting a puntatore.
3      let ptr = 0 as *mut i32;
4      unsafe {
5          // Dereferenziazione. Segmentation fault nei casi fortunati.
6          println!("{}", *ptr); // Undefined behavior
7      }
8  }
```

Per creare un puntatore è sufficiente utilizzare l'operatore di casting sul valore che vogliamo trasformare e usare il costrutto ***mut T** per creare un puntatore di tipo T (nel nostro esempio i32). Il casting di un valore a puntatore può essere eseguito normalmente all'interno del codice safe, invece se vogliamo dereferenziare un puntatore dobbiamo dichiarare un blocco unsafe e tramite l'operatore di dereferenziazione è possibile accedere al valore contenuto. Quest'ultima operazione è unsafe in quanto è possibile accedere a indirizzi di memoria arbitrari (si veda come nell'esempio stiamo provando a dereferenziare un puntatore nullo). Tipicamente la creazione di un puntatore su un indirizzo di memoria specifico non ha un vero e proprio senso, spesso l'oggetto di trasformazione in puntatore sono i riferimenti ai valori:

```
1  fn main() {
2      let mut v = 5;
3
4      // creazione di un puntatore, il riferimento viene
5      // disattivato subito dopo questo comando in quanto
6      // non è più utilizzato e non è legato al puntatore.
```

```
7       let ptr = &mut v as *mut i32;
8       unsafe {
9           println!("PTR={}", *ptr); // Dereferenziazione. Stampa: PTR=5.
10      }
11  }
```

Occorre osservare che l'utilizzo dei puntatori non è soggetto al meccanismo di ownership e borrowing e non hanno un meccanismo di verifica, quindi è possibile accedere in modo arbitrario al valore contenuto nell'indirizzo di memoria indicato dal puntatore (rimane comunque il vincolo di non poter trasferire l'ownership di un valore attraverso un puntatore). Inoltre è possibile trasformare un puntatore in un riferimento sempre utilizzando un blocco unsafe per permettere il suo utilizzo all'interno del codice safe tramite i costrutti **&*** o **&mut ***:

```
1   fn main() {
2
3       let mut v = 5;
4       let ptr = &mut v as *mut i32; // Casting a puntatore.
5       unsafe {
6           let r = &mut *ptr; // Cast da puntatore a riferimento mutabile.
7       }
8   }
```

In questo caso il puntatore è con accesso mutabile perché è dichiarato come ***mut T** con T istanziato al tipo i32, ma esistono anche puntatori con permessi di sola lettura e possono essere dichiarati con il costrutto ***const T**:

```
1   fn main() {
2       let mut v = 5;
3       let ptr_mut = &mut v as *mut i32; // Mutabile
4       let ptr_immut = &v as *const i32; // Immutabile
5       unsafe {
6           *ptr_mut = 6; // Ok! Il puntatore è con accesso mutabile.
7           println!("V: {}", *ptr_immut); // Ok! Sola lettura.
8           *ptr_immut = 7; // Errore! Il puntatore è in sola lettura.
```

```
    }
}
```

Precisiamo che a partire dai riferimenti è possibile creare solo puntatori che hanno gli stessi diritti di accesso o inferiori. Ad esempio non si può creare un puntatore mutabile se abbiamo accesso a un riferimento immutabile:

```
fn main() {
    let v = 5;
    // Errore! Il riferimento è immutabile, non è possibile
    // creare un puntatore con permessi di accesso mutabili.
    let ptr = &v as *mut i32;
    unsafe {
        *ptr = 6;
    }
}
```

Inoltre, le conversioni da puntatore a riferimento necessitano al programmatore di introdurre dei meccanismi di verifica, perché i riferimenti creati a partire da puntatori non sono validati in base alla durata del valore a cui fanno riferimento e quindi il meccanismo di ownership e borrowing non controlla che gli utilizzi siano fatti solo mentre il valore riferito è valido (quindi non protegge da un eventuale rilascio di memoria associata alla variabile e successivo utilizzo del riferimento) [16]. Per chiarire questo problema si osservi l'esempio seguente:

```
fn main() {
    let s = String::from("A");
    let ptr = &s as *const String; // Puntatore a s.
    let dangling_ref;
    unsafe {
        // Trasformazione da puntatore a riferimento a s.
        dangling_ref = &*ptr;
    }
    drop(s); // Rilascio della memoria di s.
    println!("{}", dangling_ref); // Dangling reference. Stampa: ???
}
```

Questo programma compila senza errori e avendo fatto una conversione unsafe da puntatore a riferimento, il riferimento creato è utilizzabile all'esterno di blocchi unsafe (cioè all'interno del codice safe), ma la sua durata non dipende da nessuna variabile, poiché il legame di durata viene distrutto nel momento in cui viene creato il puntatore. In questo caso abbiamo la creazione di una dangling reference, perché il riferimento è utilizzato dopo che la memoria è associata alla variabile v è stata rilasciata tramite la funzione drop. Esistono dei meccanismi che permettono di associare manualmente una durata ai puntatori quando si utilizza codice unsafe, ma verranno approfonditi nel capitolo 2.

Accesso a variabili statiche mutabili

Abbiamo visto che è possibile creare variabili globali tramite la keyword **static** ed è possibile accedervi normalmente al di fuori di blocchi unsafe, ma questo vale solo se le variabili sono dichiarate come immutabili. Nel caso in cui le variabili sono dichiarate come mutabili, poiché l'accesso in modifica è condiviso per tutto il programma, per esse non può essere regolato l'accesso tramite il meccanismo di ownership e borrowing e quindi queste variabili sono utilizzabili solo all'interno di blocchi unsafe:

```
1  static v1_immut = 1;
2  static mut v2_mut = 2;
3  fn main() {
4      println!("V1: {}", v1_immut); // Ok! Immutabile.
5      println!("V2: {}", v2_mut); // Errore! v2_mut è mutabile e statica
6
7      // Per accedere al valore è necessario usare un blocco unsafe.
8      unsafe {
9          println!("V2: {}", v2_mut); // Ok! Mutabile, ma dentro unsafe.
10     }
11 }
```

L'idea di forzare l'utilizzo dei blocchi unsafe per dereferenziare i puntatori e accedere alle variabili globali mutabili è quella di assicurarsi che non si possano verificare errori di accesso alla memoria senza che il programmatore ne sia consapevole.

Invocare funzioni unsafe

Una funzione può essere dichiarata anche unsafe e questo consente di utilizzare codice unsafe all'interno del corpo della funzione in modo analogo a un blocco unsafe. La differenza tra dichiarare una funzione unsafe rispetto a usare un blocco unsafe all'interno di una funzione sta nel metodo di invocazione, infatti una funzione dichiarata unsafe può essere invocata solo all'interno di un blocco unsafe, mentre nell'altro caso la funzione è invocabile normalmente al di fuori di blocchi unsafe:

```
1  unsafe fn null_deref() {
2      let ptr = 0 as *mut i32; // puntatore nullo
3      *ptr; // deref di un puntatore nullo
4  }
5
6  fn main() {
7      null_deref(); // Errore! Invocazione fuori da un blocco unsafe
8      unsafe {
9          null_deref(); // Ok! Invocazione dentro unsafe
10     }
11 }
```

Solitamente si dichiara una funzione unsafe quando la funzione non si occupa di garantire la safety della memoria, mentre si utilizza un blocco unsafe all'interno di una funzione quando essa si occupa internamente di garantire la safety della memoria in modo da poter essere invocata attraverso l'utilizzo di codice safe senza introdurre errori di accesso alla memoria.

Implementare traits unsafe

Quando alcuni trait definiscono alcune caratteristiche che non possono essere verificate dal compilatore, è possibile dichiararli come unsafe, in modo da impedire la loro implementazione normalmente tramite codice safe e permettere solo un implementazione forzata tramite unsafe. Precedentemente abbiamo detto che i trait Send e Sync vengono implementati dal compilatore se e solo se tutti i campi di una struttura li implementano a loro volta e non possono essere implementati manualmente, in realtà i trait sono dichiarati come unsafe e quindi non possono essere implementati esplicitamente senza codice unsafe. Questa cosa è voluta in modo da non permettere di rendere

tipi che non sono thread-safe utilizzabili in ambito multi-thread utilizzando codice safe, creando di conseguenza qualcosa che non è safe. A volte però alcuni tipi sappiamo per certo per via del loro contesto di utilizzo che sono thread-safe anche se non tutti i campi lo sono, quindi è possibile forzare l'implementazione di questi trait tramite il costrutto **unsafe impl**:

```
1  struct X<T> {
2      // I puntatori non implementano né Send né Sync
3      ptr: *mut T
4  }
5
6  unsafe impl<T> Send for X<T> { }
7  unsafe impl<T> Sync for X<T> { }
```

I puntatori sono definiti come non thread-safe (non implementano né Send, né Sync), di conseguenza nemmeno le strutture che li contengono implementeranno i traits. Per forzare l'implementazione è necessario utilizzare unsafe e il compito di verificare che l'utilizzo del tipo in ambito multi-thread non porti a nessun problema passa al programmatore. I traits unsafe (come i trait normali) possono essere anche implementati in modo condizionale rispetto all'istanziazione di uno o più generic. Nell'esempio precedente i trait vengono implementati indipendentemente dal tipo assegnato al generic T. Per implementarli in modo condizionale è sufficiente definire quali trait deve implementare il tipo a cui viene istanziato il generic oppure enumerare direttamente i tipi nel modo seguente:

```
1  struct X<T> {
2      // I puntatori non implementano né Send né Sync
3      ptr: *mut T
4  }
5  // Richiedere che il tipo T implementi un certo trait
6  unsafe impl<T: Send> Send for X<T> { }
7  unsafe impl<T: Sync> Sync for X<T> { }
8
9  // Enumerare i tipi per cui vale
10 unsafe impl Send for X<i32> { }
11 unsafe impl Sync for X<i32> { }
```

Nel primo caso imponiamo che il tipo a cui viene istanziato il generic T implementi i trait Send e/o Sync a sua volta. Nel secondo caso imponiamo che il generic T sia istanziato esattamente al tipo i32.

Riferimenti e Puntatori

Come visto nelle sezioni precedenti, il linguaggio definisce un concetto di riferimento e un concetto di puntatore ed entrambi trasportano un indirizzo di memoria, però hanno delle differenze [6]:

- I riferimenti possono essere definiti solo in funzione di un valore valido, non possono contenere un indirizzo arbitrario, sono soggetti al meccanismo di ownership e borrowing, rimangono validi al più quanto il valore riferito e possono essere creati al di fuori dei blocchi unsafe. Queste regole valgono in generale, ma fanno eccezione i riferimenti creati usando codice unsafe, convertendo un puntatore in un riferimento che in quel caso non hanno una validità precisa.

- I puntatori a differenza dei riferimenti non hanno nessuna di queste proprietà, infatti possono contenere indirizzi arbitrari, non sono soggetti al meccanismo di ownership borrowing, non hanno una durata e in più rispetto ai riferimenti è possibile effettuare casting arbitrari a altri tipi di puntatori senza restrizioni (si veda l'esempio all'inizio della sezione dove abbiamo convertito un numero i32 in un puntatore di tipo i32 con indirizzo nullo).

2. Lifetimes

I lifetimes sono il meccanismo in Rust che permette di validare la durata di un riferimento creato tramite il meccanismo di borrowing. A ogni variabile o in generale a ogni valore, al momento della sua creazione gli viene associato un lifetime.
Questo meccanismo serve per verificare le regole di ownership e borrowing, infatti per le regole di ownership un lifetime inizia al momento della creazione di una variabile e termina alla fine del blocco o al primo assegnamento (spostamento di ownership del valore contenuto). Per i riferimenti creati tramite borrowing, un lifetime inizia al momento della loro creazione e termina a partire del comando successivo dove non è più utilizzato o dal momento in cui la variabile riferita perde l'ownership sul valore. Questo lifetime viene quindi ereditato dai riferimenti che vengono creati su una variabile tramite borrowing, in modo da poter controllare a tempo di compilazione che l'utilizzo di quest'ultimi sia fatto solamente mentre il valore riferito è valido e in possesso della variabile. Nel caso di strutture e funzioni i lifetimes assumono un significato differente e assumono il nome di **generic lifetimes**.

Riferimenti a valori

Ogni volta che creiamo una variabile, il compilatore associa a essa un lifetime. Quando un valore viene spostato di owner oppure termina lo scope di definizione della variabile a cui è assegnato, automaticamente tutti i suoi riferimenti vengono invalidati e non è più possibile utilizzarli per accedere al valore. Per chiarire questo concetto, analizziamo un esempio dove viene mostrata la proprietà di invalidazione dei riferimenti:

```
1  fn invalid_lifetime() {
2      // I lifetimes si indicano con un'apostrofo prima del nome
3      let r;                // ---------+-- 'a
4      {                     //          |
5          let x = 5;        // -+-- 'b  |
6          r = &x;           //  |       |
7      }                     // -+       |
```

```
8       println!("R={}", r);  //           |
9   }                         // ---------+
```

In questo caso l'errore del compilatore è il seguente:

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

L'errore ci avverte che il valore di *x* è stato liberato al termine del suo scope e noi stiamo provando ad accedervi da uno scope più esterno attraverso un riferimento che non è più valido. Per ottenere una versione valida è sufficiente aumentare il lifetime della variabile *x* spostandola nello scope più esterno in questo modo:

```
1   fn valid_lifetime() {
2       let x = 5;              // ----------+-- 'a
3       let r;                  // --+-- 'a  |
4       {                       //   |       |
5           r = &x;             //   |       |
6       }                       //   |       |
7       println!("R={}", r);     //   |       |
8                               // --+       |
9   }                           // ----------+
```

A questo punto il compilatore non restituisce nessun errore, perché il riferimento alla variabile *x* ha un lifetime uguale a quello di *r*.

Riferimenti nelle funzioni

L'utilizzo dei lifetimes nelle funzioni assume un significato diverso rispetto a quello sulle variabili e sui riferimenti, infatti in questo caso non servono per dare una validità a un riferimento in base alla definizione dell'owner del valore, ma servono per definire una relazione tra i lifetimes dei riferimenti passati come parametri e eventualmente tra i lifetimes dei riferimenti restituiti. Ad

esempio, in questo caso il riferimento ritornato viene validato con lo stesso lifetime associato al parametro da cui dipende:

```
1  fn valid_return_lifetime(word: &mut String) -> &str {
2      word.push('!');
3      &word[0..3]
4  }
```

Da questo esempio in realtà possiamo notare anche un'altra regola molto importante, cioè che è possibile ritornare riferimenti immutabili a partire da riferimenti mutabili. A questo punto potremmo pensare che sia possibile invocare più volte questo metodo mentre il riferimento ritornato è ancora attivo, in realtà non è possibile, perché da come notiamo dal corpo della funzione, una successiva invocazione porterebbe a una modifica della stringa mentre un riferimento immutabile è ancora attivo e questo porterebbe a una violazione della regola AXM. Rust in questi casi possiamo dire che crea una catena di dipendenze di riferimenti e mette in sospeso la disattivazione dei riferimenti finché tutti i riferimenti creati a partire da esso non vengono disattivati. Nel nostro esempio, Rust crea una dipendenza tra il parametro e il valore di ritorno e non disattiva il riferimento mutabile passato come parametro finché il riferimento immutabile rimane attivo, impedendo un'altra invocazione del metodo data dalla necessità di creare un nuovo riferimento mutabile. Vediamo un esempio dove si può vedere questa creazione di catena di dipendenze:

```
1  fn main () {
2      let mut s = String::from("Hello");
3      let slice1 = valid_return_lifetime(&mut s); // Prima invocazione
4      let slice2 = valid_return_lifetime(&mut s); // Seconda invocazione
5
6      // Riferimento della prima invocazione utilizzato
7      println!("Slice1: {}", slice1);
8  }
```

In questo caso l'errore del compilatore è il seguente:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
  --> src\main.rs:11:40
   |
10 |     let slice1 = valid_return_lifetime(&mut s); // Prima invocazione
   |                                         ------ first mutable borrow occurs
   here
11 |     let slice2 = valid_return_lifetime(&mut s); // Seconda invocazione
   |                                         ^^^^^^ second mutable borrow
   occurs here
...
14 |     println!("Slice1: {}", slice1);
   |                            ------ first borrow later used here
```

Il motivo di questo errore è dato dal fatto che il riferimento restituito *slice1* viene utilizzato dopo la seconda invocazione e per via della catena di dipendenze il primo riferimento mutabile non può essere disattivato e quindi il compilatore ci avvisa che non è possibile creare un secondo riferimento mutabile.

Un altro caso particolare è quando il riferimento ritornato dipende da più parametri, poiché sono disponibili più combinazioni, Rust chiede di scrivere esplicitamente la relazione che c'è tra i riferimenti in ingresso e quelli in uscita. Vediamo un esempio di funzione dove il riferimento ritornato dipende da più riferimenti in ingresso:

```
1  fn unknown_return_lifetime(w1: &String, w2: &String,
2                             take: bool) -> &String {
3      if take {
4          w1
5      } else {
6          w2
7      }
8  }
```

In questo caso l'errore del compilatore è il seguente:

```
error[E0106]: missing lifetime specifier
 --> src\main.rs:2:43
  |
1 | fn unknown_return_lifetime(w1: &String, w2: &String,
  |                                -------      -------
2 |                            take: bool) -> &String {
  |                                          ^ expected named lifetime parameter
  |
```

Il compilatore ci avvisa che non riesce a capire che relazione c'è tra questi riferimenti e ci chiede di definire noi una relazione valida. Per definire una relazione tra i lifetimes, dobbiamo dichiarare dei generic lifetimes dopo il nome della funzione e dobbiamo associarli ai riferimenti usando il costrutto **&'**. Ad esempio, possiamo definire una relazione che impone che tutti i riferimenti abbiano lo stesso lifetime nel modo seguente:

```
1  fn known_return_lifetime<'a>(w1: &'a String, w2: &'a String,
2                               take: bool) -> &'a String {
3      if take {
4          w1
5      } else {
6          w2
7      }
8  }
```

Con questa forma di dichiarazione di lifetimes, in realtà non si richiede che i riferimenti abbiano la stessa durata, ma si richiede che i lifetime dei riferimenti passati come parametro siano accorciabili a un lifetime comune e che il riferimento ritornato abbia validità pari a quella del riferimento che vive per meno tempo (equivale al lifetime comune a entrambi). Questo succede perché il lifetime **'a** dei parametri compare in posizioni covarianti e quindi permette di diminuire la durata di un riferimento. Il funzionamento completo lo approfondiremo quando parleremo di varianza nella sezione successiva e per il momento ci limitiamo a dire che devono essere uguali.

Riferimenti nelle strutture

Nelle strutture, il concetto di lifetimes si applica sia per specificare la relazione che esiste tra i riferimenti presenti in essa, sia per dare un tempo di validità alla struttura stessa. Vediamo un esempio di struttura che contiene due riferimenti a un tipo generico T:

```
struct RefHolder<'a, T> {
    r1: &'a T,
    r2: &'a T
}
```

In questo caso stiamo chiedendo che entrambi i lifetime dei riferimenti devono essere accorciabili a uno unico in comune a entrambi (per via della covarianza) e che la struttura viene invalidata e non può più essere utilizzata non appena il riferimento più breve non è più valido. Ad esempio, se i valori sono dichiarati in scope differenti:

```
fn main() {
    let holder;
    let n1 = 0;
    {
        let n2 = 1;
        // Per via della covarianza la durata del riferimento a
        // n1 viene accorciata a quella del riferimento a n2.
        holder = RefHolder {
            r1: &n1,
            r2: &n2
        }
    }

    // La struttura non più utilizzabile, il campo r2 non è più valido.
    holder.r1;
}
```

non è più possibile utilizzare la struttura per accedere ai campi dopo la fine dello scope più interno e infatti il compilatore ci avvisa che l'accesso al campo non è più valido mediante l'errore seguente:

```
error[E0597]: `n2` does not live long enough
  --> src\main.rs:16:17
   |
16 |             r2: &n2
   |                 ^^^ borrowed value does not live long enough
17 |         }
18 |     }
   |     - `n2` dropped here while still borrowed
...
21 |     holder.r1;
   |     --------- borrow later used here
```

Lifetimes speciali

Nel linguaggio sono presenti due lifetimes con un significato speciale:

- Il lifetime **'static** che serve per indicare che il riferimento ha una durata pari all'intero programma e può essere utilizzato solo per referenziare funzioni, string literals (stringhe statiche) e variabili definite nello scope globale. Questo lifetime è sempre maggiore o uguale di tutti gli altri.

- Il lifetime anonimo '_ è un lifetime speciale che aiuta nelle definizioni dei metodi e blocchi implementazione quando non ci interessa il valore del lifetime in una determinata posizione. Questo lifetime viene interpretato dal compilatore come se il programmatore dichiarasse ogni volta un nuovo generic lifetime.

Quando un generic lifetime compare solo in una posizione assume un valore jolly e implica che accetta qualsiasi lifetime in ingresso. Questo succede perché non sono presenti dei vincoli tra più lifetime, quindi il generic lifetime può assumere direttamente il valore necessario.

2.1 Varianza

Il linguaggio non definisce un concetto di ereditarietà tra strutture, quindi non è presente nemmeno il concetto di varianza per i generics anche se abbiamo visto che sono presenti nel linguaggio (Capitolo 1). Rust invece sposta questo concetto di varianza sui lifetimes [21] per poter gestire in modo più flessibile il loro utilizzo permettendo di utilizzare un lifetime che ha una durata diversa in un contesto in cui il suo utilizzo non porterebbe a nessun errore. I lifetime assumono una varianza diversa a seconda della posizione di utilizzo all'interno delle dichiarazioni dei riferimenti e di

dividono in: covarianti, contravarianti e invarianti. Precisiamo che per questi concetti si sono consultati principalmente il libro di Rust e il Rustonomicon [16] che purtroppo è incompleto e molti degli aspetti trattati (come i lifetimes) non sono esaustivi e ben documentati.

Lifetime covariante

```
1  // Funzione che prende in ingresso due riferimenti in sola lettura.
2  fn covariant<'a>(w1: &'a str, w2: &'a str) {
3      println!("w1: {}, w2: {}", w1, w2);
4  }
5
6  fn main() {
7      let short = String::new();
8
9      // Riferimento con validità pari all'intero programma.
10     let long_ref: &'static str = "Hello";
11     // Riferimento con validità pari al corpo della funzione main.
12     let short_ref: &str = &short;
13     covariant(long_ref, &short_ref);
14 }
```

Un lifetime covariante [16] permette di diminuire a piacimento la durata di un riferimento, permettendo di inserire all'interno di una struttura o passare come parametro, riferimenti con un tempo di validità maggiore o uguale a quello richiesto. Nel nostro esempio [21], la funzione prende in ingresso due riferimenti immutabili a stringhe e nel nostro caso stiamo provando a invocarla con una stringa che ha un tempo di validità pari a **'static** e un'altra che ha come validità solamente il corpo della funzione main (**'short**). Questa cosa è corretta e non porterebbe a nessun errore dato che stiamo accorciando il lifetime **'static** al lifetime associato al riferimento alla variabile short per motivi di sola lettura.

Lifetime contravariante

```rust
// Funzione che prende in ingresso sia un riferimento mutabile a un
// riferimento a funzione che richiede come parametro un
// riferimento con lifetime 'static, sia un'altra funzione
// che richiede un riferimento in ingresso con lifetime 'short.
fn contravariant<'short>(f1: &mut fn(&'static str) -> (),
    f2: fn(&'short str) -> ()) {

    // 'static è sempre maggiore o uguale a ogni altro lifetime
    // quindi 'short sarà sempre minore o uguale e quindi è
    // perfettamente corretto inserire nella locazione riferita
    // un riferimento a una funzione che permette di gestire
    // parametri che hanno una durata minore a quella richiesta.
    *f1 = f2;
}
```

Un lifetime contravariante [16] permette di aumentare a piacimento la durata di un riferimento, permettendo di inserire all'interno di una variabile o passare come parametro, riferimenti con un tempo di validità minore o uguale a quello richiesto. Nel linguaggio l'unica fonte di contravarianza è quando utilizziamo funzioni higher-order, infatti questa varianza viene inferita solo se il lifetime è utilizzato nei parametri della funzione passata come parametro a una funzione (o campo di una struttura). Nel nostro esempio [21], stiamo provando a inserire un riferimento a una funzione che accetta un lifetime **'short** in ingresso dentro un riferimento a una funzione che richiede un lifetime **'static** in ingresso, quindi è perfettamente corretto inserire dentro quest'ultimo un riferimento a una funzione che può gestire un riferimento con durata inferiore a quanto aspettato, tanto ci verrà fornito un argomento con durata superiore rispetto a quanto possiamo gestire. Il contrario non sarebbe corretto (cioè rendere il lifetime covariante) altrimenti la nostra funzione si aspetterebbe di operare su qualcosa che ha una durata maggiore rispetto a quanto in realtà vale.

Lifetime invariante

```rust
// Funzione che prende in ingresso sia riferimento mutabile a un
// riferimento a un valore di tipo stringa, sia un riferimento a
```

```
3  // un valore di tipo stringa. Notare che l'argomento del riferimento
4  // mutabile non può essere un riferimento a funzione, altrimenti i
5  // lifetime dei parametri sarebbero contravarianti per quanto già visto.
6  fn invariant<'a, 'b>(s: &'b mut &'a str, x : &'a str) {
7      *s = x;
8  }
9
10 fn main() {
11     let mut literal1: &'static str = "A";
12     let literal2: &'static str = "B";
13     invariant(&mut literal1, &literal2);
14     println!("Literal1={}", literal1); // Stampa: Literal1=B
15 }
```

Un lifetime di tipo invariante [16] non permette, né di aumentare, né di diminuire la durata di un riferimento, permettendo di operare solo con altri riferimenti che hanno esattamente la stessa durata. Questo tipo di varianza tipicamente viene assegnata nel caso in cui abbiamo accesso mutabile a un riferimento a un valore (doppia indirezione). Nel nostro esempio [21] stiamo provando a inserire dentro una variabile che possiede un riferimento con validità **'static** un altro riferimento con lo stesso lifetime. Potrebbe sembrare strano e non corretto imporre questa invarianza in questo caso, in realtà è necessaria e per spiegare il perché useremo un controesempio [21]:

```
1  // Precisiamo che il lifetime invariante è il secondo lifetime del
2  // primo parametro (cioé &'a str). Non è vera questa cosa per il
3  // lifetime assegnato 'b (&'b mut). L'invarianza vale per l'argomento
4  // del riferimento mutabile NON per il riferimento mutabile stesso.
5  fn invariant<'a, 'b>(s: &'b mut &'a str, x: &'a str) {
6      *s = x;
7  }
8
9  fn main() {
10     let mut literal: &'static str = "Hello";
11     let local = String::new();
```

```
12      let local_ref: &str = &local;
13
14      // Sovrascrittura del riferimento
15      invariant(&mut literal, &local_ref);
16      drop(local); // drop del valore associato a local
17      println!("{}", literal); // Dangling?
18  }
```

Assumiamo per assurdo che questo codice compili senza problemi. Se il lifetime non fosse invariante noi potremmo diminuire la durata del riferimento literal (**'static**) a quella del riferimento local_ref (**'short**) nel caso di covarianza. A questo punto noi potremmo inserire dentro la variabile literal un riferimento alla variabile local e successivamente liberare la memoria di local senza che il riferimento literal venga invalidato (perché crede di essere in possesso di un riferimento a una stringa con validità pari all'intera durata del programma). A questo punto se provassimo a leggere il valore del riferimento literal andremmo a leggere una zona di memoria non coerente con quanto ci aspettiamo e anche invalida a causa dell'operazione di drop ottenendo come risultato una dangling reference. Lo stesso ragionamento può essere applicato con la contravarianza, perché anche in quel caso possiamo illudere l'owner del riferimento di dipendere da un determinato lifetime, mentre in realtà dipende da un altro e ogni informazione su questa dipendenza sarebbe distrutta in modo analogo per via della contravarianza. L'unico modo per evitare questo problema è di operare con riferimenti che hanno esattamente la stessa durata e per questo motivo i riferimenti mutabili sono invarianti **rispetto al tipo a cui fanno riferimento** (nel nostro esempio un riferimento a una stringa).

Precisiamo per evitare ambiguità che questa dichiarazione è corretta:

```
1  fn invariant<'a, 'b>(s: &'b mut &'a str, x: &'a str) {
2      *s = x;
3  }
4
5  fn main() {
6      let mut literal: &'static str = "Hello";
7      let local = String::new();
```

```
8       let mut local_ref: &str = &local;
9       invariant(&mut local_ref, &literal); // Ok, parametri scambiati.
10      drop(local);
11      println!("{}", literal); // Ok! È ancora valido.
12      println!("{}", local_ref); // Errore! Non è più valido.
13  }
```

Il motivo per cui questa dichiarazione è valida non è perché il lifetime del primo parametro è contravariante, ma perché il lifetime nel secondo parametro compare in posizione covariante, quindi la durata del secondo riferimento può essere diminuita fino a quella del primo parametro che richiede un lifetime specifico per via della sua invarianza. Se il secondo parametro a sua volta fosse invariante, questa invocazione non sarebbe possibile e il programma non compilerebbe.

Di seguito riportiamo una tabella riassuntiva che mostra le varie varianze in vari casi di utilizzo nel linguaggio [16]:

		'a	T	U
*	`&'a T`	covariant	covariant	
*	`&'a mut T`	covariant	invariant	
*	`Box<T>`		covariant	
	`Vec<T>`		covariant	
*	`UnsafeCell<T>`		invariant	
	`Cell<T>`		invariant	
*	`fn(T) -> U`		**contra**variant	covariant
	`*const T`		covariant	
	`*mut T`		invariant	

Notiamo che anche alcune strutture speciali chiamati smart pointers che vedremo nel prossimo capitolo (Box, Vec) hanno una varianza associata, il motivo è analogo a quanto spiegato prima e anche in questo caso dipende dalla posizione di utilizzo del lifetime per la struttura nei campi delle strutture. Nella tabella sono inoltre utilizzati dei tipi parametrici in alcune dichiarazioni. Questo perché il concetto di varianza non si applica solo con dei casi specifici e vale per ogni tipo. Precisia-

mo che per tipo in questo caso non intendiamo i tipi normali come i32, ma intendiamo riferimenti a tipi, perché la varianza viene applicata solo sui lifetime dei riferimenti. Ad esempio, è possibile creare un vettore che contiene riferimenti a interi, istanziandolo nel modo seguente:

```
1  // Funzione che prende in ingresso un riferimento a un vettore
2  // e un altra stringa con lo stesso lifetime come parametro.
3  fn concat_first<'short>(vector: &Vec<&'short str>,
4          s : &'short str) -> String {
5      // Concatena la prima stringa del vettore a quella passata
6      // come parametro e restituisci la stringa risultato.
7      let mut concat = String::from(vector[0]);
8      concat.push_str(s);
9      concat
10 }
11 fn main() {
12     // Vettore di interi
13     let mut vector: Vec<&'static str> = Vec::new();
14
15     // Stringa con validità 'static
16     let long_ref: &'static str = "A";
17     vector.push(long_ref);
18
19     // Stringa con validità 'short (corpo del main)
20     let short_string = String::from("B");
21     let short_ref: &str = &short_string;
22
23     let result = concat_first(&vector, short_ref);
24     println!("Concat={}", result); // Stampa Concat=AB
25
26 }
```

In questo caso i riferimenti all'interno del vettore hanno una validità **'static** e il riferimento alla stringa passata come secondo parametro ha un lifetime **'short** (pari al corpo del main). In questo

caso, dato che Vec è covariante rispetto al suo lifetime (e non è presente accesso mutabile tramite &mut) il lifetime **'static** può essere accorciato fino a quello del secondo parametro in modo da permettere l'invocazione della funzione. Per il caso di invarianza (&mut T) nell'esempio precedente equivale a istanziare T con **&'b i32** (cioè **&mut &'b i32**), in questo modo il lifetime **'b** sarà invariante rispetto alla durata istanziata.

Precisiamo inoltre, che nel caso di strutture il lifetime assume una singola varianza che dipende dalle posizioni in cui compare. Ad esempio nella struttura seguente:

```
1  struct ConflictLifetime<'a, 'b> {
2      r1: &'a String, // Ok, 'a dovrebbe essere covariante,
3      r2: &'b mut &'a String // ma compare anche in posizione invariante
4  }
```

il lifetime **'a** compare in posizioni con varianza diversa, quindi l'invarianza vince il conflitto e nel caso in cui la struttura viene ad esempio passata come parametro a una funzione:

```
1  fn conflict<'a,'b>(conf_struct1: &'b mut ConflictLifetime<'a, 'b>,
2          conf_struct2: &'b ConflictLifetime<'a, 'b>) {
3      conf_struct1.r1 = conf_struct2.r1;
4  }
```

il lifetime **'a** della struttura sarà invariante, quindi non è possibile aumentare o diminuire la durata dei riferimenti e le strutture passate alla funzione devono avere esattamente la stessa durata (per il lifetime **'a**).

2.2 Utilizzi avanzati

Nel linguaggio sono presenti anche altri meccanismi che permettono di manipolare e utilizzare i lifetime in contesti diversi rispetto al loro scopo di base.

PhantomData

A volte potremmo ritrovarci in una situazione dove vogliamo inserire un lifetime con una determinata varianza e una specifica durata all'interno di una struttura, in modo da poter regolare gli

utilizzi della struttura ove non vi siano vincoli di per sé (tipicamente dovuti all'utilizzo di codice unsafe). Il problema è che il compilatore non ci lascia definire un generic lifetime all'interno delle strutture se non viene utilizzato (perché non riesce a capire a cosa serve e non riesce nemmeno a inferirgli una varianza adeguata), perciò il programmatore è costretto a definire un campo aggiuntivo che utilizza un lifetime anche se nella definizione non viene mai utilizzato (portando a uno spreco di memoria). Il progetto di Rust prevede la presenza di questo tipo di situazione, infatti il compilatore per risolvere questo problema mette a disposizione del programmatore il tipo speciale *PhantomData* [8, 33], che consente di definire dei campi fantasma, cioè campi che esistono a tempo di compilazione e seguono tutte le regole imposte dal compilatore, ma al momento dell'ottimizzazione vengono eliminati ovunque compare il loro utilizzo. Ad esempio, con questo tipo possiamo dichiarare un lifetime con la varianza che desideriamo (seguendo le regole descritte dalla tabella della sezione 2.1) nel seguente modo:

```
1  struct CovariantLifetime<'id>(PhantomData<&'id ()>);
2  struct InvariantLifetime<'id>(PhantomData<*mut &'id ()>);
3  struct ContravariantLifetime<'id>(PhantomData<fn(&'id ()) -> ()>);
```

In questo modo è possibile utilizzarli nelle definizioni delle strutture come campi dove vogliamo in modo da imporre la varianza desiderata su un lifetime, perché tanto hanno dimensione zero e quindi vengono eliminate dai campi delle strutture. Ad esempio se vogliamo, rendere due strutture invarianti in modo che solo le strutture con lo stesso lifetime possano essere passate a una funzione, possiamo dichiarare una struttura nel modo seguente:

```
1  struct MyStruct<'a> {
2      val: i32,
3      _marker: InvariantLifetime<'a>
4  }
5
6  fn sum_same_lifetime<'a>(s1: &MyStruct<'a>, s2: &MyStruct<'a>) {
7      println!("Sum: {}", s1.val + s2.val)
8  }
```

La definizione del generic lifetime **'a** per la funzione e l'utilizzo per mettere in relazione le strutture che contengono lifetime invarianti, impone il vincolo che entrambe le strutture devono avere esat-

tamente lo stesso lifetime. In altre parole non è possibile passare una struttura che ha un lifetime **'l1** e un'altra che ha un lifetime **'l2** con **'l1** diverso da **'l2**. Precisiamo che PhantomData permette di definire una varianza ma non crea automaticamente una relazione con altri lifetime per assegnare una specifica durata ed è compito del programmatore inizializzare il lifetime mettendo in relazione il lifetime creato tramite PhatomData con un altro già esistente che possiede una durata precisa.

Validazione dei puntatori

Abbiamo visto che i lifetimes creati tramite PhantomData possono servire a vincolare l'utilizzo delle strutture che contengono determinati valori in determinati contesti, in realtà questo approccio può essere applicato anche nel caso in cui le strutture contengono dei puntatori, che per definizione, contengono un indirizzo di memoria arbitrario e quindi non possiedono un lifetime che ne indica la durata. Nel secondo caso il programmatore che utilizza codice unsafe, può evitare di limitarne l'utilizzo solo all'interno del tipo che sta definendo e può servirsi dei lifetime definiti manualmente, per rilasciare in sicurezza all'esterno questi puntatori definendo una relazione tra la struttura principale e una struttura di supporto contenente un puntatore per un utilizzo esterno. Sfruttando la varianza e la possibilità di legare la durata della struttura che contiene il puntatore alla struttura a cui si riferisce, è possibile invalidare il puntatore quando la struttura a cui fa riferimento viene eliminata. Ad esempio, se vogliamo accedere in modo mutabile a una locazione da più owner differenti usando codice unsafe garantendo l'invalidazione automatica di tutti i puntatori quando la locazione viene rilasciata potremmo utilizzare la strategia seguente:

```
1  // Il corpo dei metodi è stato nascosto e non ci interessa,
2  // in quanto vogliamo solo mostrare come entrano in relazione
3  // i lifetime delle strutture per validare i vari puntatori.
4  struct Loc<'a, T> {
5      // Smart pointer speciale per violare le regole di
6      // ownership e borrowing all'interno di strutture
7      loc: UnsafeCell<T>,
8      _marker: CovariantLifetime<'a>
9  }
10
11 struct LocRef<'a, T> {
```

```
12      ptr: *mut T,
13      _marker: CovariantLifetime<'a>
14  }
15
16  // PhatomData si inizializza passando il nome come parametro
17  impl<'a, T> Loc<'a, T> {
18      fn new_ref(&'a self) -> LocRef<'a, T> {
19          LocRef {
20              ptr: self.loc.get(),
21              _marker: CovariantLifetime(PhantomData)
22          }
23      }
24  }
25
26  impl<'a, T> LocRef<'a, T> {
27      fn get<'b>(&'b self) -> &'b T {
28          unsafe {
29              &(*self.ptr)
30          }
31      }
32
33      fn get_mut<'b>(&'b mut self) -> &'b mut T {
34          unsafe {
35              &mut (*self.ptr)
36          }
37      }
38  }
39
40  fn main() {
41      // Assumiamo che sia definito un metodo new sulla struttura Loc.
42      let loc = Loc::new(1);
43      let mut ref1 = loc.new_ref();
```

```
44      let mut ref2 = loc.new_ref();
45      *ref1.get_mut() = 2; // Ok! Riferimento valido.
46      *ref2.get_mut() = 3; // Ok! Riferimento valido.
47      drop(loc); // Locazione non più valida.
48      *ref1.get_mut() = 4; // Errore! Riferimento non più valido.
49  }
```

Nella definizione della struttura Loc è stato utilizzato il tipo UnsafeCell, questo tipo verrà approfondito nel capitolo 3, quindi per il momento ci limitiamo a dire che è un tipo speciale che permette di ottenere un puntatore con accesso mutabile al valore contenuto anche se l'intera struttura è accessibile tramite riferimento immutabile, cosa non normalmente possibile per quanto visto nel capitolo 1.

Notiamo che grazie ai lifetime definiti manualmente siamo riusciti a validare i puntatori contenuti nelle strutture mettendo in relazione i lifetime delle strutture ritornate dal metodo *new_ref* (RefLoc) con il lifetime della struttura che contiene la vera locazione di memoria (Loc). Questo non è l'unico modo di collegare dei lifetime per imporre dei vincoli sull'utilizzo di una struttura, infatti esistono moltissimi altri contesti, ad esempio quando si utilizzano i lifetime invarianti per vincolare l'utilizzo delle strutture si parla di **branded types**.

Branded types

I branded types [33] sono una particolare applicazione dei lifetime invarianti, che sfrutta le chiusure offerte dal linguaggio per assegnare alle istanze delle strutture un identificatore univoco all'interno del codice (*brand*), in modo da vincolare l'utilizzo di una particolare istanza su altre strutture, impedendo di operare con altre istanze con lifetime diverso (brand diverso). Per analizzare il loro funzionamento, utilizziamo un esempio che mostra come sia possibile accedere a un elemento di un vettore tramite indice senza che venga effettuato il controllo sulla lunghezza del vettore, avendo la garanzia che l'indice acceduto sia all'interno del vettore [33]:

```
1  // Vedremo Vec nel capitolo dedicato agli smart pointer e per
2  // adesso ci limitiamo a dire che è un array ridimensionabile
3  struct BrandedVec<'a> {
4      vec: Vec<i32>,
```

```
5       _brand: InvariantLifetime<'a>
6   }
7
8   struct BrandedIndex<'a> {
9       index: i32,
10      _brand: InvariantLifetime<'a>
11  }
12
13  impl<'a> BrandedVec<'a> {
14      // Invoca la chiusura passata come parametro con una
15      // nuova istanza del vettore con un nuovo brand (lifetime).
16      fn new(f: impl for<'brand>
17              FnOnce(BrandedVec<'brand>)) {
18          f(BrandedVec { /* ... */})
19      }
20
21      // Ritorna il valore a partire dall'indice passato
22      // come parametro. Il lifetime 'a si riferisce
23      // al lifetime istanziato sull'istanza stessa.
24      fn get(&self, idx: &BrandedIndex<'a>) -> i32 {
25          self.vec.get_unchecked(idx.index) // senza bound check
26      }
27
28      // Aggiungi il valore dentro al vettore e ritorna
29      // un indice alla posizione del valore aggiunto.
30      // Il lifetime 'a si riferisce a quello
31      // istanziato sull'istanza stessa.
32      fn add(&mut self, val: T) -> BrandedIndex<'a> {
33          self.vec.push(val);
34          BrandedIndex {
35              index: self.vec.len() - 1,
36              _brand: InvariantLifetime(Default::default())
```

```
37          }
38      }
39  }
```

In questo esempio la struttura BrandedVec definisce un metodo *new* che prende in ingresso una chiusura che assegna un nuovo lifetime (brand) al vettore passato come argomento e la invoca creando una nuova istanza della struttura. Imponendo di passare il vettore alla chiusura, imponiamo il vincolo di non poter creare più di una istanza della struttura per ogni chiusura e di conseguenza vietiamo la possibilità di avere più istanze della struttura BrandedVec che hanno lo stesso lifetime, poiché a ogni invocazione del metodo *new* viene assegnato un nuovo lifetime alla nuova istanza, sempre diverso dai precedenti e non compatibile per via della invarianza. Questa cosa implica, che se due strutture contengono lifetimes invarianti e per essere passate come parametro a un metodo questi lifetimes devono essere in relazione, in automatico garantiamo che solo le istanze delle strutture restituite dai metodi invocati su una particolare istanza possono essere passate in ingresso a altri metodi della stessa istanza per poterci operare. Ad esempio, se abbiamo creato due istanze di BrandedVec nel modo seguente [33]:

```
1  fn main() {
2      BrandedVec::new(|brand_vec1| {
3          let i1 = brand_vec1.add(1);
4
5          BrandedVec::new(|brand_vec2|{
6              let j1 = brand_vec2.add(1);
7              let j2 = brand_vec2.add(2);
8
9              // Ok! L'indice passato è stato generato
10             // dalla stessa istanza di BrandedVec
11             brand_vec1.get(&i1);
12             // Errore! Questo indice non è stato generato
13             // dalla stessa istanza di BrandedVec
14             brand_vec1.get(&j1);
15         })
```

```
16      })
17  }
```

se proviamo a accedere a qualche elemento del vettore, il compilatore grazie ai lifetime invarianti, ci assicura che il metodo *get* dell'istanza di BrandedVec sia invocato esattamente con un BrandedIndex che ha esattamente lo stesso lifetime (quindi solo con i BrandedIndex restituiti dal metodo add sulla stessa istanza) e vieta il passaggio di altri indici generati da altri vettori che potrebbero essere non validi (perché potrebbero avere dimensioni diverse), poiché il lifetime è diverso. Precisiamo che questa cosa vale solo se il vettore è monotono crescente, altrimenti se fosse possibile accorciarne la sua lunghezza, alcuni indici potrebbero non essere più validi, ma nel nostro caso non abbiamo questo problema dato che Vec non riduce la dimensione se non è espressamente richiesto dal programmatore utilizzando il metodo *resize*.

3. Smart Pointers

Rust impone delle regole molto forti per garantire l'assenza di errori di accesso alla memoria che devono essere verificate a tempo di compilazione e per via di questi limiti, molti programmi corretti non possono essere scritti normalmente in questo linguaggio. Rust per risolvere questi limiti in modo da continuare a garantire la memory safety, mette a disposizione dei programmatori alcuni tipi speciali e alcune operazioni aggiuntive (tramite l'utilizzo di codice unsafe) per definire delle strutture in grado di risolvere un problema specifico in modo efficiente violando le regole imposte dal linguaggio. Queste strutture prendono il nome di smart pointers, perché violano internamente le regole imposte dal linguaggio a tempo di compilazione, ma espongono comunque una API che garantisce il rispetto di tutte le regole a tempo di esecuzione. Queste strutture non sono esclusive di Rust, infatti hanno origine nel linguaggio C++. Questo tipo di strutture si chiamano smart pointers, perché permettono di manipolare secondo una determinata logica una zona di memoria di cui non si può normalmente dimostrare la memory safety a tempo di compilazione tramite le normali regole imposte dal linguaggio. Inoltre, essi gestiscono automaticamente anche il rilascio delle risorse quando lo smart pointer non è più utilizzato, rilasciando la memoria associata senza che il programmatore effettui una richiesta esplicita. Queste strutture in Rust oltre a permettere di gestire una zona di memoria, devono anche garantire che il loro utilizzo all'intero di codice safe (cioè la API offerta) non possa portare a violare le regole di ownership e borrowing o in generale devono garantire che il loro utilizzo non possa portare alla creazione di errori di accesso memoria come la creazione di dangling references. Solitamente (non obbligatoriamente) gli smart pointers gestiscono una zona di memoria allocata nello heap in cui non vi è una regola precisa di allocazione e deallocazione imposta, ma vedremo alcuni smart pointers che non effettuano allocazioni nello heap e si limitano a regolare gli accessi al dato inserito al loro interno.

Principalmente gli smart pointers si dividono in tre categorie, che possiamo identificare come segue:

- **Statici**: Sono smart pointers che si limitano a cambiare il comportamento del compilatore in determinate situazioni e sfruttano le regole imposte dal meccanismo di ownership e

borrowing per gestire in maniera differente l'accesso a una zona di memoria.

- **Dinamici**: Sono smart pointers che permettono di violare le regole imposte da Rust a tempo di compilazione e eseguono controlli a tempo di esecuzione per verificare la correttezza del loro utilizzo.
- **Ibridi**: Sono smart pointers che per minimizzare i costi a tempo di esecuzione, sfruttano l'analizzatore statico per verificare una parte della definizione a tempo di compilazione e utilizzano alcuni controlli a tempo di esecuzione per verificare la parte non verificabile staticamente o per offrire una maggiore flessibilità di utilizzo.

Box

In Rust è possibile gestire l'allocazione e deallocazione di una zona di memoria come in altri linguaggi come C/C++ tramite richieste esplicite al memory allocator, però non è la scelta consigliata dal linguaggio per allocare un dato nello heap, tanto che l'accesso esplicito al memory allocator è unsafe e non può essere fatto utilizzando codice safe. Per allocare nello heap esiste la struttura Box, che è uno smart pointer speciale che permette di allocare un dato nello heap senza costi aggiuntivi, perché il compilatore cancella tutte le sue occorrenze e le sostituisce con chiamate dirette al memory allocator. Il vantaggio dell'utilizzo di questa struttura rispetto a una richiesta esplicita è che la allocazione avviene al momento della creazione di un'istanza di Box e la deallocazione avviene alla distruzione dell'istanza seguendo le normali regole di gestione della memoria imposte dal RAII [13], mentre la gestione esplicita (manuale) non segue nessuna regola di allocazione e deallocazione.

Tipi ricorsivi

Gli smart pointers possono essere utilizzati in vari contesti e un possibile uso è nella definizione di tipi di dimensione variabile o ricorsivi. Questo perché senza l'utilizzo di uno smart pointer che alloca il valore nello heap, una struttura ricorsiva avrebbe una dimensione infinita. Ad esempio, se proviamo a implementare le liste concatenate singole senza nessuno smart pointer nel modo seguente:

```
1  struct Node<T> {
2      val: T,
```

```
3       next: Option<Node<T>>
4   }
```

otteniamo un errore di questo tipo:

```
error[E0072]: recursive type `Node` has infinite size
 --> src\main.rs:4:1
  |
4 | struct Node<T> {
  | ^^^^^^^^^^^^^^
5 |     val: T,
6 |     next: Option<Node<T>>
  |                  ------- recursive without indirection
```

L'errore ci avvisa che il tipo ha una dimensione infinita e ci suggerisce di aggiungere un indirezione per allocare il successore nello heap anziché sullo stack. Per risolvere questo problema, possiamo utilizzare lo smart pointer Box visto prima per imporre che il nodo successore sia allocato nello heap:

```
1   struct Node<T> {
2       val: T,
3       next: Option<Box<Node<T>>>
4   }
```

Adesso il tipo non ha più dimensione infinita, dato che Box alloca sullo stack (o in generale nella zona di memoria dedicata alla struttura Node) solo l'indirizzo di memoria al nodo successivo.

Potremmo pensare che si possono utilizzare semplicemente i riferimenti per indicare eventualmente il nodo successore e/o il predecessore, in realtà non si può perché non possiamo creare più accessi mutabili contemporaneamente e quindi nel caso in cui è necessario riferire lo stesso nodo da più nodi contemporaneamente, l'accesso alla struttura può avvenire solo in modo immutabile condividendo vari riferimenti immutabili a un nodo con altri nodi. Inoltre tutti i nodi devono avere la stessa durata, altrimenti parte della struttura ricorsiva potrebbe diventare invalida durante il suo utilizzo:

```
1   // Ok! Le liste concatenate singole si possono implementare tramite
2   // riferimenti mutabili, ma tutti i nodi devono avere la
```

```
3  // stessa durata per evitare che parte della struttura
4  // diventi invalida rilasciando la memoria dei nodi.
5  struct Node<'a, T> {
6      val: T,
7      next: Option<&'a mut Node<'a, T>>
8  }
9
10 // Ok! si può fare ma per condividere l'accesso è necessario
11 // condividere  un riferimento immutabile, quindi la struttura
12 // diventa accessibile solo in lettura, inoltre anche in questo
13 // caso tutti i nodi devono avere la stessa durata.
14 struct DoubleNode<'a, T> {
15     val: T,
16     next: Option<&'a DoubleNode<'a, T>>,
17     prev: Option<&'a DoubleNode<'a, T>>
18 }
```

Per via del meccanismo di ownership e borrowing, quindi alcuni tipi ricorsivi che richiedono ownership multipla con accesso mutabile su un nodo della struttura come le liste concatenate doppie non sono normalmente implementabili in modo efficiente. Per risolvere questo problema senza utilizzare codice unsafe direttamente, è necessario utilizzare degli smart pointers che permettono di violare questi vincoli internamente a tempo di compilazione e che nel caso multi-thread offrono anche un meccanismo di sincronizzazione. Precisiamo che nel caso di utilizzo multi-thread (come detto nel capitolo 1) i campi di una struttura devono implementare anche Send e/o Sync a loro volta affinché l'intera struttura li implementi rispettivamente. Nel caso di tipi ricorsivi questo vale solo se ogni nodo della struttura li implementa a loro volta, quindi non è sufficiente utilizzare un meccanismo di sincronizzazione sul primo nodo della struttura ricorsiva affinché la struttura ricorsiva diventi thread-safe, ma è necessario che ogni nodo della struttura lo utilizzi (causando costi a tempo di esecuzione considerevoli).

3.1 I traits per smart pointers

I traits Deref e Drop

Gli smart pointers sono strutture e quindi non è possibile normalmente trattarli come puntatori e dereferenziarli per accedere al dato contenuto. In questi casi dovremmo definire dei metodi specifici per ottenere un riferimento al dato contenuto e invocarli come se fossero funzioni. Per risolvere questo problema, Rust definisce due trait speciali chiamati Deref e DerefMut che possono essere implementati dalle strutture e permettono di utilizzare l'operatore di dereferenziazione per i puntatori sulla struttura che li implementa per accedere a un riferimento a un campo della struttura. Ad esempio se vogliamo poter accedere a un campo 'val' di una struttura X con l'operatore di dereferenziazione con accesso mutabile o immutabile possiamo implementare i trait nel modo seguente:

```
1  struct X<K> {
2      val: K
3  }
4
5  impl<K> Deref for X<K> {
6      // Questa informazione è richiesta dal trait Deref per condividere
7      // il tipo tra entrambi i trait (Deref e DerefMut).
8      type Target = K;
9
10     fn deref(&self) -> &K {
11         &self.val
12     }
13 }
14
15 impl<K> DerefMut for X<K> {
16     // K deve essere lo stesso tipo usato in Deref. Il vincolo è
17     // imposto dal costrutto 'type Target = K' che permette di
18     // vincolare il tipo utilizzato in un metodo in funzione
19     // del tipo istanziato in altri traits implementati sulla
```

```
20      // stessa struttura (o in generale sullo stesso tipo).
21      fn deref_mut(&mut self) -> &mut K {
22          &mut self.val
23      }
24  }
```

I trait Deref e DerefMut definiscono un metodo ciascuno che prende in ingresso rispettivamente un riferimento immutabile o mutabile alla struttura stessa e restituisce rispettivamente un riferimento immutabile o mutabile a un campo contenuto nella struttura.
Una volta che una struttura implementa i trait è possibile dereferenziarla con l'operatore di dereferenziazione nel modo seguente:

```
1   fn main() {
2       let mut smrt_ptr = X {
3           val: 0
4       }
5
6       // Dereferenziazione con accesso mutabile.
7       *smrt_ptr = 1;
8       // Dereferenziazione con accesso immutabile.
9       println!("Smart Ptr={}", *smrt_ptr); // Stampa Smart Ptr=1
10  }
```

Inoltre, gli smart pointers a volte possono contenere dei puntatori internamente e di conseguenza la memoria associata non verrebbe rilasciata automaticamente come imposto dal RAII come succede invece normalmente con scalari, stringhe o valori allocati nello heap tramite Box. Per risolvere questo problema, Rust definisce un altro trait speciale chiamato Drop che permette di definire un metodo distruttore che permette di eseguire del codice personalizzato quando lo smart pointer (o in generale una struttura) termina il suo scope e viene richiesto il rilascio della memoria associata. Ad esempio, se vogliamo rilasciare la memoria associata a un puntatore quando una struttura viene distrutta, possiamo implementare il trait Drop nel modo seguente:

```
1   struct X {
2       ptr: *mut i32
```

```
}

impl X {
    fn new(v: i32) -> Self {
        Self {
            // Alloca nello heap, distruggi il box e restituisci
            // un puntatore alla zona allocata in modo da gestire
            // manualmente tramite codice unsafe la zona di memoria.
            ptr: Box::into_raw(Box::new(0))
        }
    }
}

impl Drop for X {
    fn drop(&mut self) {
        unsafe {
            // Ricostruisci il box a partire da un puntatore.
            let own = Box::from_raw(self.ptr);
            // Distruggi il Box e rilascia la memoria.
            drop(own);
        }
    }
}
```

Quando una struttura implementa il trait Drop, prima di rilasciarne la memoria associata, il compilatore invoca il metodo drop definito su di essa per permettere di eseguire del codice personalizzato per rilasciare la memoria dei campi che non sono gestiti automaticamente.
Precisiamo che il metodo drop non può essere invocato manualmente, infatti se lo invochiamo esplicitamente:

```
let smrt_ptr = X::new(0);
smrt_ptr.drop();
```

otteniamo un errore di questo tipo:

```
error[E0040]: explicit use of destructor method
  --> src\main.rs:31:14
   |
31 |     smrt_ptr.drop();
   |     ---------^^^^-- 
   |     |        |
   |     |        explicit destructor calls not allowed
```

Questo accade perché il compilatore invoca il metodo drop (distruttore) sempre e automaticamente al momento del rilascio della memoria, quindi se fosse possibile effettuare anche una chiamata esplicita a questo metodo si avrebbe un errore di tipo double free.

I traits Send e Sync

Esistono smart pointers che non possono essere usati in ambito multi-thread perché evitano il meccanismo delle lock per aumentare le performance e altri che sfruttano questo meccanismo per garantire l'assenza di race conditions e quindi possono essere usati. Come visto nel capitolo 1, un tipo per essere utilizzato in ambito multi-thread deve implementare i trait Send e/o Sync, quindi anche per gli smart pointer vale la stessa regola. Il trait Send permette di inviare lo smart pointer tramite passaggio di ownership a un thread, mentre il trait Sync permette di condividere l'accesso allo smart pointer a tanti thread diversi allo stesso tempo attraverso un riferimento immutabile. Quasi tutti i tipi nel linguaggio implementano Send, ma esistono delle eccezioni e tipicamente riguardano gli smart pointers che permettono di avere ownership multipla su un valore senza meccanismi di sincronizzazione come ad esempio Rc che analizzeremo tra poco. Il trait Sync anche in questo caso è implementato da quasi tutti i tipi, ma esistono alcune eccezioni e tipicamente riguardano i tipi che permettono di avere interior mutability (accesso mutabile attraverso riferimento immutabile) senza meccanismi di sincronizzazione come RefCell che analizzeremo tra poco.

Nel caso di strutture che sono smart pointers, poiché solitamente utilizzano dei puntatori per violare internamente alcune regole imposte dal linguaggio, essi di base non possono implementare i traits Send e Sync, in quanto l'utilizzo di un puntatore non è controllato dal compilatore. In questi casi è possibile implementarli forzatamente mediante un implementazione unsafe e il compito di verificare che non possano verificarsi errori di accesso alle memoria spetta al programmatore.

3.2 Analisi di smart pointer predefiniti

Smart pointers per collezioni

Rust fornisce smart pointers come ad esempio Vec e LinkedList per operare su sequenze di dati di dimensione variabile:

```
1  let mut vec = Vec::new();
2  let mut list = LinkedList::new();
3
4  // Aggiungi un elemento in coda al vettore.
5  vec.push(1);
6  // Rimuovi l'elemento in coda al vettore.
7  vec.pop();
8
9  // Aggiungi un elemento in coda alla lista.
10 list.push_back(1);
11 // Rmuovi un elemento in testa alla lista.
12 list.pop_front();
```

La differenza tra questi due smart pointers è che Vec utilizza un array ridimensionabile per contenere i valori inseriti, mentre LinkedList utilizza nodi doppiamente collegati. Inoltre, questi due smart pointer hanno il vantaggio di offrire accesso mutabile e immutabile a più valori all'interno di essi contemporaneamente tramite uno slice:

```
1  let mut vec = Vec::from([1, 2, 3]);
2  // Slice con accesso mutabile ai primi tre valori.
3  let mut slice = &mut vec[0..2];
4  // Imposta i primi due valori dello slice a 0.
5  slice[0] = 0;
6  slice[1] = 0;
```

Sono presenti anche smart pointers come ad esempio HashSet e HashMap per operare su insiemi di dati senza duplicati di dimensione variabile:

```rust
let mut map = HashMap::new();
let mut set = HashSet::new();

// Se non è già presente una mappatura per la stringa
// "Hello" aggiungi all'interno della mappa l'associazione
// altrimenti sovrascrivi il valore precedente associato a "Hello".
map.insert("Hello", 1);
// La rimozione si effettua passando per
// riferimento la chiave da rimuovere.
map.remove(&"Hello");

// Aggiungi valore intero all'interno del set se non è
// presente altrimenti sovrascrivi il valore precedente.
set.replace(1);
// La rimozione si effettua passando per
// riferimento il valore da rimuovere.
set.remove(&1);

// Nel caso di rimozione per mappe e per sets, se il valore /
// associazione non è presente, il metodo remove non produce
// nessun effetto (non modifica la collezione di valori).
```

La cosa interessante di tutti questi smart pointers è che sono thread safe e tutti quanti implementano i trait Send e Sync se i rispettivi tipi inseriti dentro di essi li implementano. Questo vale perché non permettono di avere ownership multipla e per modificare i dati contenuti dentro di essi è necessario un accesso mutabile allo smart pointer, quindi l'accesso è regolato direttamente dal meccanismo di ownership e borrowing anche per l'ambito multi-thread.

UnsafeCell

Rust definisce uno smart pointer speciale per l'utilizzo tramite codice unsafe che si chiama UnsafeCell. Questo smart pointer definisce solo tre metodi:

- Un metodo *new* che permette di creare lo smart pointer con all'interno un dato.

- Un metodo *get_mut* che permette di ottenere un puntatore mutabile al valore contenuto.
- Un metodo *into_inner* che permette di distruggere la cella e recuperare il valore contenuto.

Questo smart pointer è speciale per il compilatore, perché ogni volta che compare tra i campi di una struttura, lo cancella e lascia al suo posto solamente il dato contenuto in esso. Questa cosa permette di violare le regole imposte dal compilatore per accedere al dato tramite l'utilizzo dei puntatori all'interno di un blocco unsafe (usando codice unsafe) indipendentemente dalla presenza di accesso mutabile o no sulla struttura che lo contiene senza pagare costi aggiuntivi in termini di memoria e tempo.

RefCell e Mutex

RefCell è uno smart pointer che offre interior mutability, cioè permette di accedere in lettura e scrittura al dato contenuto garantendo la memory safety anche quando lo smart pointer è condiviso tramite riferimenti immutabili. Questo smart pointer permette quindi di violare la regola di AXM a tempo di compilazione, ma comunque verifica che sia rispettata a tempo di esecuzione controllando gli accessi attivi sul dato nella cella. Nell'esempio che segue mostriamo come sia possibile modificare il valore contenuto mentre la cella è condivisa in lettura per poi stamparne il contenuto:

```
1  fn main() {
2      let cell = RefCell::new(0);
3      let l1 = &cell;
4      let l2 = &cell;
5
6      // Creiamo un riferimento mutabile al valore.
7      let ref_mut = l1.borrow_mut();
8      *ref_mut = 1;
9
10     // Eliminiamo il riferimento mutabile altrimenti la
11     // prossima chiamata causerebbe un errore fatale a tempo
12     // di esecuzione per garantire il rispetto della regola
13     // AXM e il thread verrebbe immediatamente terminato.
14     drop(ref_mut);
```

```
15
16      // Creiamo un riferimento immutabile al valore
17      let ref_immut = l2.borrow();
18      println!({}, *ref_immut); // Stampa 1
19  }
```

Questo smart pointer implementa Send se il dato contenuto a sua volta lo implementa, in quanto l'invio a un altro thread non introduce race conditions o altri errori, ma non implementa Sync in nessun caso, perché i metodi *borrow* e *borrow_mut* richiedono un riferimento immutabile alla cella per accedere al dato e ai contatori interni per tenere traccia dei riferimenti attivi in scrittura senza meccanismi di sincronizzazione. Questa cosa in caso di condivisione tra più threads causerebbe la possibilità che si verifichino delle race conditions che portano alla violazione della regola AXM. Per un utilizzo multi-thread esiste lo smart pointer Mutex, che offre le stesse funzionalità di Refcell, ma utilizza una lock interna per garantire che solo un thread alla volta possa accedere al dato contenuto:

```
1   static cell = RefCell::new(0);
2   static mutex = Mutex::new(0);
3   fn main() {
4       for i in 0..10 {
5           // Errore! RefCell non implementa Sync non
6           // può essere catturato tramite riferimento.
7           thread::spawn(|| {
8               let r = cell.borrow_mut();
9               *r += 1;
10              println!({}, *r);
11          });
12      }
13
14      // Se rimuoviamo il codice precedente ...
15
16      for i in 0..10 {
17          // Ok! Mutex implementa Sync, perché utilizza il
```

```
18          // meccanismo delle lock per regolare gli accessi,
19          // quindi va bene catturarlo tramite riferimento.
20          thread::spawn(|| {
21              let guard = mutex.lock().unwrap();
22              *guard += 1;
23              println!(Guard={}, *guard); // Stampa 10 volte: Guard=1
24          });
25      }
26  }
```

I metodi *borrow* di Refcell e *lock* di Mutex restituiscono rispettivamente una Ref e una LockGuard per fare riferimento al valore contenuto dentro la rispettiva cella con i permessi di accesso acquisiti. Questi smart pointers utilizzano anche dei lifetimes internamente, infatti i riferimenti restituiti sono legati alla durata della cella stessa e vengono invalidati automaticamente quando la memoria associata al valore viene rilasciata:

```
1   fn main() {
2       let cell = RefCell::new(5);
3       let r = cell.borrow_mut();
4
5       *r = 0; // Ok! Riferimento sempre valido.
6       drop(cell);
7       *r = 1; // Errore! Riferimento non più valido.
8
9       /* Discorso analogo per Mutex. */
10  }
```

Infine, consultando la documentazione [2] notiamo che RefCell e Mutex non compaiono tra gli smart pointers che allocano il dato contenuto nello heap, infatti sono delle eccezioni che salvano il dato come campo della struttura nella stessa zona di memoria dove la struttura viene allocata (eventualmente anche sullo stack) e si limitano a regolare gli accessi al dato tramite metodi e strutture di appoggio.

Rc e Arc

Rc è uno smart pointer che implementa un reference counter e permette di condividere l'ownership su di un valore. Per evitare di violare la regola AXM, questo smart pointer permette di accedere al valore contenuto solo in lettura tramite riferimenti immutabili. Inoltre, questo smart pointer ci garantisce che la memoria associata al valore non verrà liberata finché sono presenti ancora degli accessi strong (forti) a quel valore. Per generare un nuovo riferimento strong a partire da un altro dobbiamo utilizzare il metodo clone nel modo seguente:

```
1  fn main() {
2      let rc1 = Rc::new(0); // Crea il reference counter.
3
4      {
5          let rc2 = Rc::clone(&rc1); // Copia il reference counter
6          // ... altre operazioni
7      } // rc2 viene distrutto
8      // ... alter operazioni
9  } // rc1 viene distrutto ed essendo l'ultimo riferimento
10 // anche la memoria associata al valore viene rilasciata.
```

L'utilizzo dei reference counter può causare dei leak di memoria se si creano dei cicli tra strutture contenenti altri reference counters e se non si eliminano manualmente, quindi per risolvere questo problema e evitare di preoccuparsi di gestire i cicli, questo smart pointer definisce anche un weak reference counter, che è un reference counter analogo a quello appena descritto, ma che non influisce sul conteggio dei riferimenti. Per creare un weak reference counter dobbiamo utilizzare il metodo downgrade offerto dallo smart pointer e passare lo strong reference counter che ci interessa 'retrocedere':

```
1  fn main() {
2      let rc1 = Rc::new(0);
3      let rc2 = Rc::downgrade(&rc1);
4  }
```

Per accedere al valore tramite un weak reference counter, visto che non influisce sul conteggio dei riferimenti attivi e quindi la memoria associata potrebbe essere stata rilasciata, dobbiamo chiedere

una ‘promozione’ tramite il metodo upgrade che ci restituisce uno strong reference counter se la memoria è ancora valida oppure niente se non lo è più:

```
fn main() {
    let strong_rc1 = Rc::new(0);
    let weak_rc2 = Rc::downgrade(&strong_rc1);

    // L'ultimo riferimento strong eliminato, la memoria
    // del reference counter viene rilasciata.
    drop(strong_rc1);

    match weak_rc2.upgrade() {
        None => { /* Non si può usare */ }
        Some(strong_rc2) => {
            /* Si può usare e accedere tramite strong_rc */
        }
    }

    // In questo caso verrà eseguito il ramo None.
}
```

In questo modo è possibile creare cicli all’interno delle strutture senza dover definire un algoritmo che elimina manualmente tutti i cicli creati quando la struttura viene distrutta.
Questo smart pointer non implementa né Send, né Sync, in quanto la clonazione dello smart pointer può essere fatta avendo accesso a un riferimento immutabile al reference counter tramite il metodo *clone* e il trasferimento dell’ownership di un reference counter a un altro thread permetterebbe al nuovo thread di incrementare e decrementare il contatore interno in contemporanea ad altri threads senza meccanismi di sincronizzazione.
Per l’utilizzo anche in ambito multi-thread esiste lo smart pointer Arc che utilizza una lock per aggiornare il contatore interno:

```
fn main() {
    let rc1 = Rc::new(1);
```

```
3       for i in 0..10 {
4           let rc2 = Rc::clone(&rc1);
5           thread::spawn(move || {
6               println("{}", rc2); // Errore Rc non implementa Send
7           });
8       }
9
10      let arc1 = Arc::new(1);
11      for i in 0..10 {
12          let arc2 = Arc::clone(&arc1);
13          thread::spawn(move || {
14              println("{}", arc2); // Ok Arc implementa Send
15      });
16
17      println("{}", arc1);
18  }
```

Come per il reference counter Rc, anche Arc definisce una versione weak che non influisce sul conteggio dei riferimenti attivi per l'utilizzo multi-thread:

```
1   fn main() {
2       let strong_arc1 = Arc::new(1);
3       let weak_arc1 = Arc::downgrade(&strong_arc1);
4       thread::spawn(move || {
5           match weak_arc1.upgrade() {
6           None => { /* Memoria non più valida */ }
7           Some(strong_arc2) => {
8               /* Memoria ancora valida */
9               }
10          }
11      });
12
```

```
13      println("{}", strong_arc1);
14  }
```

Precisiamo che questi smart pointer per l'utilizzo multi-thread non permettono di rendere thread safe qualsiasi tipo, infatti implementano i trait Send e/o Sync se e solo se il tipo contenuto li implementa a loro volta, questo perché ad esempio inserire un reference counter (Rc) dentro un Mutex non vieta di clonarlo per continuare a utilizzarlo anche dopo aver rilasciato la lock per accedere al valore e continuare a alterare il contatore interno attraverso thread diversi. Se infatti consultiamo la documentazione vedremo che Mutex e Arc implementano i trait in modo condizionale:

```
1  // Attenzione! La definizione reale non è esattamente questa!
2  // Dai vincoli sul generic nelle definizioni è stato rimosso
3  // il trait ?Sized dato che non è necessario per i nostri scopi.
4
5  impl<T: Send + Sync> Send for Arc<T> {}
6  impl<T: Send + Sync> Sync for Arc<T> {}
7
8  impl<T: Send> Send for Mutex<T> {}
9  impl<T: Send> Sync for Mutex<T> {}
```

Notiamo che Arc non offrendo alcun meccanismo aggiuntivo, oltre a garantire l'aggiornamento atomico del contatore interno, implementa i trait Send e Sync se e solo se il tipo contenuto a sua volta implementa sia Send che Sync. Mutex invece, permette di condividere l'accesso mutabile tra più thread al valore contenuto tramite l'utilizzo di una lock, quindi richiede solamente che il tipo contenuto sia inviabile a un altro thread e che quindi implementi solo Send.

4. Un caso di studio: GhostCell

GhostCell [33] è uno smart pointer che si distingue da tutti gli altri, perché utilizza in modo originale e completamente diverso il sistema di analisi statica dei lifetimes e il concetto di allocazione su arene [4, 22] per garantire accessi immutabili e mutabili tramite un token di autorizzazione per dimostrare possesso o meno su un particolare dato a tempo di compilazione anche in presenza di accesso condiviso. Un'arena è un tipo di memory allocator, che permette di allocare valori singolarmente su richiesta, mentre non permette la deallocazione dei valori singoli e la memoria utilizzata viene rilasciata solo quando l'intera arena viene distrutta. GhostCell definisce una API rigorosa, che permette di definire tipi ricorsivi senza usare codice unsafe (quindi puntatori) e senza pagare alcuna operazione aggiuntiva a tempo di esecuzione. In particolare, lo smart pointer sfrutta la varianza dei lifetimes che abbiamo approfondito nel capitolo 2 e l'ottimizzazione del compilatore per:

- Evitare costi aggiuntivi a tempo di esecuzione per verificare la memory safety con l'aiuto di alcuni tipi speciali per ignorare internamente le regole imposte dal meccanismo di ownership e borrowing e forzare manualmente il comportamento del compilatore.
- Verificare che l'accesso al dato contenuto nello smart pointer rispetti la regola AXM a tempo di compilazione spostando i controlli effettuati dal meccanismo di ownership e borrowing sul token di accesso anziché sullo smart pointer diretto.

La motivazione e il vantaggio che ha spinto alla definizione di questo smart pointer, è la possibilità di definire tipi ricorsivi utilizzando esclusivamente codice safe senza pagare alcun operazione aggiuntiva a tempo di esecuzione, rispetto a definire la stessa struttura ricorsiva utilizzando codice unsafe (cioè i puntatori). Come spiegato nel capitolo dedicato 3, per via della natura 'fail-safe' del compilatore, è molto costoso in termini di operazioni aggiuntive o a volte impossibile definire tipi ricorsivi utilizzando Rust nella sua versione pura (cioè senza smart pointers e puntatori). Spesso quindi dobbiamo utilizzare uno dei due approcci e in entrambi i casi avremo vantaggi e svantaggi. Nel caso in cui utilizziamo i puntatori attraverso codice unsafe, pur avendo molta più libertà,

dobbiamo dimostrare esplicitamente che quello che stiamo facendo non violi le regole imposte dal linguaggio e questo può essere molto oneroso in termini di tempo. Nel caso in cui utilizziamo gli smart pointers dinamici dobbiamo scegliere con cura quello più adatto al problema, perché pur non dovendo dimostrare la correttezza del programma, avremo un costo in operazioni aggiuntive a tempo di esecuzione per garantire la correttezza dell'utilizzo. Nel caso in cui utilizziamo smart pointer statici 'general purpose' per evitare alcune operazioni aggiuntive, avremo dei vincoli di flessibilità molto forti nella definizione (e probabilmente non tutti i tipi ricorsivi sarebbero implementabili) non essendo specializzati per questo particolare problema. Come possiamo capire, in tutte le strategie abbiamo degli svantaggi non indifferenti, GhostCell invece consente di affrontare il problema riducendo gli svantaggi sfruttando il sistema di analisi statica dei lifetimes per definire tipi ricorsivi utilizzando solamente i riferimenti (quindi solo codice safe). Come vedremo, lo smart pointer si limita a usare il sistema di analisi statica del compilatore e non esegue nessuna operazione aggiuntiva a tempo di esecuzione, quindi possiamo dire che GhostCell rientra nella categoria degli smart pointers statici.

4.1 Implementazione dello smart pointer GhostCell

GhostCell è implementato usando i seguenti tipi [23]:

```
1  struct InvariantLifetime<'id>(PhantomData<*mut &'id ()>);
2
3  struct GhostToken<'id> {
4      _marker: InvariantLifetime<'id>,
5  }
6
7  struct GhostCell<'id, T> {
8      _marker: InvariantLifetime<'id>,
9      value: UnsafeCell<T>,
10 }
```

InvariantLifetime

```
1       struct InvariantLifetime<'id>(PhantomData<*mut &'id ()>);
```

La struttura InvariantLifetime [23], sfrutta quanto detto nel capitolo dedicato ai lifetimes per dare un brand alle strutture (GhostToken e GhostCell) senza occupare memoria aggiuntiva a tempo di esecuzione. La struttura infatti utilizza al suo interno solamente un PhantomData per forzare la varianza del lifetime **'id** come invariante e poiché PhantomData viene cancellato dai campi della struttura durante l'ottimizzazione, anche la struttura viene eliminata ovunque compare il suo utilizzo avendo dimensione pari a zero.

GhostToken

```
1  struct GhostToken<'id> {
2      _marker: InvariantLifetime<'id>,
3  }
4
5  impl<'id> GhostToken<'id> {
6      fn new<R>(
7          f: impl for<'new_id> FnOnce(GhostToken<'new_id>) -> R) -> R {
8          f(GhostToken {
9              _marker: InvariantLifetime(PhantomData)
10         })
11     }
12 }
```

La struttura GhostToken [23] contiene solamente un campo di tipo InvariantLifetime, il cui ruolo è dare il brand alla struttura senza occupare memoria aggiuntiva per quanto detto sopra. Questa struttura da sola non permette di sfruttare il brand, perché permetterebbe di creare più GhostToken aventi lo stesso lifetime. Per evitare questo problema, la struttura non specifica un metodo *new* che ritorna un'istanza di GhostToken nel suo blocco implementazione, bensì il metodo prende in ingresso una chiusura che verrà invocata con un'istanza di GhostToken a cui verrà assegnato un lifetime nuovo e non compatibile con nessun altro per via della sua invarianza grazie al costrutto **impl for<'new_id>** [7, 16] che permette di generare un lifetime in funzione della chiusura passata

come parametro, permettendo di istanziare un nuovo lifetime 'fresco' a ogni invocazione della chiusura e non in funzione del corpo del metodo *new* come invece succederebbe se usassimo il costrutto **new<'new_id>**. Le scritture sono diverse e se usassimo la seconda, il brand al token non verrebbe assegnato correttamente, dato che il lifetime non sarebbe in relazione con niente all'interno del corpo del metodo.

GhostCell

```
1  struct GhostCell<'id, T> {
2      value: UnsafeCell<T>,
3      _marker: InvariantLifetime<'id>
4  }
5
6  impl<'id, T> GhostCell<'id, T> {
7      fn new(value: T) -> Self {
8          Self {
9              _marker: InvariantLifetime(PhantomData),
10             value: UnsafeCell::new(value)
11         }
12     }
13
14     fn into_inner(self) -> T {
15         unsafe { self.value.into_inner() }
16     }
17
18     fn get_mut(&mut self) -> &mut T {
19         unsafe { &mut *self.value.get() }
20     }
21
22     fn borrow<'a>(&'a self, token: &'a GhostToken<'id>) -> &'a T {
23         unsafe { &*self.value.get() }
24     }
```

```
25
26      fn borrow_mut<'a>(&'a self,
27          token: &'a mut GhostToken<'id>) -> &'a mut T {
28          unsafe { &mut *self.value.get() }
29      }
30  }
```

La struttura GhostCell [23] contiene un campo di tipo UnsafeCell per contenere il dato vero e proprio e un campo di tipo InvariantLifetime il cui scopo è quello di definire il brand della struttura e permettere l'associazione di un solo token, che per via della sua invarianza e per via dello specifico utilizzo delle chiusure da parte dello smart pointer, impone sempre l'utilizzo della stessa istanza dopo la prima occorrenza di utilizzo su un'istanza della struttura (struttura GhostCell). Questa struttura definisce anche un blocco implementazione dove troviamo cinque metodi principali:

- I metodi *new* e *into_inner* per rispettivamente creare una GhostCell a partire da un valore e consumare un'istanza per ottenere il valore contenuto in essa nel caso in cui siamo owner della cella.
- Il metodo *get_mut* che permette di ottenere un riferimento mutabile al valore contenuto nella cella nel caso in cui il chiamante abbia direttamente accesso esclusivo alla cella.
- I metodo *borrow* e *borrow_mut*, che permettono di ottenere rispettivamente un riferimento immutabile o mutabile al valore contenuto nella cella quando non si ha accesso esclusivo alla cella stessa. Entrambi i metodi per evitare di violare la regola AXM imposta da Rust richiedono il passaggio per riferimento del token di autorizzazione associato alla cella. Passare solamente il token come parametro non è sufficiente, perché in quel modo sarebbe possibile invocare i metodi più volte senza restrizioni e questo permetterebbe di avere dei riferimenti non compatibili attivi allo stesso momento. Per evitare questo problema è necessario definire un generic lifetime **'a** e vincolare la durata del riferimento al token a quella del riferimento ritornato. In questo modo non è possibile creare altri riferimenti al token non compatibili con la regola AXM mentre è possibile accedere al contenuto di un nodo della struttura ricorsiva.

Separazione dei dati dai permessi

Come abbiamo visto, lo smart pointer è definito da una struttura che fa da contenitore dei dati (GhostCell) e un'altra che regola l'accesso (GhostToken). Questa definizione permette di modellare il concetto di separazione dei dati dai permessi [33], perché il dato è contenuto in una GhostCell che può essere acceduta in modo mutabile anche in presenza di riferimenti immutabili grazie al GhostToken che a seconda del tipo di borrow effettuato regola l'accesso alla cella (o alla struttura ricorsiva se stiamo implementando tipi ricorsivi) e garantisce che la regola AXM non sia violata.

4.2 Realizzazione di liste con e senza GhostCell

In questa sezione presenteremo l'implementazione delle liste concatenate doppie tramite Ghostcell e tramite gli smart pointers predefiniti in Rust sia nell'ambito single-thread, sia nell'ambito multi-thread. In particolare evidenzieremo le differenze a livello definizione per mostrare i vantaggi e gli svantaggi offerti dall'utilizzo dei vari smart pointer.

Liste single-thread, usando Rc e RefCell

Una possibile definizione di liste concatenate doppie per l'utilizzo single-thread è la seguente:

```
1  struct SafeNode<T> {
2      val: T,
3      next: Option<Rc<RefCell<SafeNode<T>>>>,
4      prev: Option<Weak<RefCell<SafeNode<T>>>>,
5  }
6
7  impl<T> SafeNode<T> {
8      // Metodo costruttore per la lista (testa della lista).
9      fn new(item: T) -> Rc<RefCell<SafeNode<T>>> {
10         Rc::new(RefCell::new(SafeNode {
11             val: item,
12             prev: None,
13             next: None,
14         }))
```

```
15      }
16  }
```

Per l'ambito single-thread per implementare le liste concatenate doppie, abbiamo bisogno di utilizzare un reference counter per violare il vincolo di un solo owner per ogni valore, in modo da poter avere più owner per lo stesso nodo. Inoltre, Rc permette di accedere al valore contenuto solo in lettura tramite un riferimento immutabile per non violare la regola AXM per via della presenza di molteplici owner. A causa di questo vincolo abbiamo bisogno anche di una cella che offre interior mutability, cioè che permette di accedere in modo mutabile al nodo anche se abbiamo accesso a solamente un riferimento immutabile (fornito dal reference counter) e quindi abbiamo bisogno di utilizzare lo smart pointer RefCell. Notiamo che nella definizione per il nodo predecessore invece di Rc, è stato utilizzato Weak, che come spiegato nel capitolo 3, è un reference counter che non influisce sul contatore dei riferimenti attivi ai nodi, in modo da non dover eliminare tutti i cicli presenti in tutti i nodi manualmente quando viene richiesto il rilascio della memoria associata alla lista.

Il metodo per aggiungere un nodo con valore 'v' dopo la idx-esima posizione può essere implementato nel modo seguente:

```
1  impl<T> SafeNode<T> {
2      // n: reference counter alla testa della lista.
3      // v: valore da aggiungere.
4      // idx: posizione desiderata dove inserire come successore.
5      pub fn add_next(n: Rc<RefCell<SafeNode<T>>>, v: T, idx: u32) {
6          let mut curr = n;
7          let mut cp;
8          let mut index = 0;
9
10         // Scorri la lista fino all'indice desiderato
11         // oppure fino all'ultimo nodo se l'indice va
12         // oltre la lunghezza della lista.
13         while index < idx {
14             match curr.borrow().next {
15                 None => {
```

```
16                      break;
17                  }
18                  Some(ref next) => {
19                      cp = Rc::clone(next);
20                      index += 1;
21                  }
22              }
23              curr = cp;
24          }
25
26          // Crea un nuovo nodo
27          let val = SafeNode::new(v);
28          val.borrow_mut().prev = Some(Rc::downgrade(&curr));
29
30          // Aggiorna i campi del nodo predecessore e successore
31          // inserendo come nodo intermedio quello appena creato
32          let mut curr_borrowed = curr.borrow_mut();
33          match curr_borrowed.next {
34              None => {}
35              Some(ref node) => {
36                  val.borrow_mut().next = Some(Rc::clone(node));
37                  node.borrow_mut().prev = Some(Rc::downgrade(&val));
38              }
39          }
40
41          curr_borrowed.next = Some(Rc::clone(&val));
42      }
43  }
```

Per iniziare, dobbiamo scorrere la lista fino a trovare il nodo a cui vogliamo modificare il successore per inserire un nuovo valore e questo può essere fatto con un loop, con un indice per sapere a quale nodo siamo arrivati e sfruttando il metodo borrow offerto da RefCell per accedere al nodo

in lettura. Notiamo che ogni volta che accediamo al successore, siccome non siamo autorizzati a spostare l'ownership visto che abbiamo accesso tramite riferimento, dobbiamo clonare il reference counter e salvarlo in una variabile per l'iterazione successiva. Una volta arrivati al nodo desiderato, dobbiamo cambiare i valori dei successori e dei predecessori dei nodi coinvolti nell'operazione grazie al metodo *borrow_mut* di RefCell che permette di accedere in modifica anche se abbiamo solo accesso in lettura per via del reference counter.

Il metodo per rimuovere un elemento nella lista è molto simile al metodo precedente e può essere implementato nel modo seguente:

```
1  impl<T> SafeNode<T> {
2        // n: reference counter alla testa della lista.
3        // idx: posizione da rimuovere. Il metodo non gestisce
4        // il caso limite di rimozione della testa, quindi
5        // l'indice passato deve essere maggiore di 0.
6        pub fn remove_at(n: Rc<RefCell<SafeNode<T>>>,
7           idx: u32) -> Option<T> {
8           let mut curr = n;
9           let mut cp;
10          let mut index = 0;
11
12          // Scorri la lista fino all'indice desiderato
13          // oppure fino all'ultimo nodo se l'indice va
14          // oltre la lunghezza della lista
15          while index < idx {
16              match curr.borrow().next {
17                  None => {
18                      break;
19                  }
20                  Some(ref next) => {
21                      cp = Rc::clone(next);
22                      index += 1;
```

```
23                  }
24              }
25              curr = cp;
26          }
27
28          // Nel caso di indice 0 non fare niente (testa della lista)
29          if index == 0 {
30              return None;
31          }
32
33          // Salva il nodo da rimuovere
34          let to_remove = curr.borrow();
35
36          // Aggiorna i campi del nodo predecessore.
37          match to_remove.prev {
38              None => {}
39              Some(ref node) => {
40                  match to_remove.next {
41                      None => {
42                          node.upgrade().unwrap().borrow_mut().next =
43                              None;
44                      }
45                      Some(ref next) => {
46                          node.upgrade().unwrap().borrow_mut().next =
47                              Some(Rc::clone(next));
48                      }
49                  }
50              }
51          }
52
53          // Aggiorna i campi del nodo successore.
54          match to_remove.next {
```

```
55              None => {}
56              Some(ref node) => {
57                  match to_remove.prev {
58                      None => {
59                          node.borrow_mut().prev = None;
60                      }
61                      Some(ref prev) => {
62                          node.borrow_mut().prev =
63                              Some(Rc::clone(prev));
64                      }
65                  }
66              }
67          }
68
69          // Distruggi gli altri reference counter in modo da
70          // avere solo uno strong reference counter attivo.
71          drop(to_remove);
72          // Ritorna il valore contenuto nel nodo.
73          Some(Rc::try_unwrap(curr).ok().unwrap().into_inner().val)
74      }
75  }
```

Anche in questo caso scorriamo la lista fino a trovare il nodo desiderato e poi tramite Refcell accediamo in modo mutabile per modificare i predecessori e i successori del nodo da rimuovere. Per accedere il predecessore di un nodo dobbiamo chiedere una promozione del weak reference counter a un reference counter normale, perché essendo che non altera il contatore dei riferimenti attivi, la memoria del reference counter potrebbe essere stata rilasciata (non vale in questo caso ma in generale potrebbe).

Il metodo per iterare su una lista può essere realizzato scorrendo la lista nel modo seguente:

```
1  impl<T> SafeNode<T> {
2      // head: riferimento alla testa della lista.
```

```
3       // token: riferimento immutabile al token associato alla lista.
4       // f: chiusura da invocare su ogni nodo durante l'iterazione.
5       pub fn iter(head: Rc<RefCell<SafeNode<T>>>,
6           mut f: impl FnMut(&T)) {
7           let mut curr = head;
8           let mut cp;
9           loop {
10              // Scorri la lista e su ogni nodo visitato
11              // invoca la chiusura passata come parametro.
12              let to_iter = curr.borrow();
13              match to_iter.next {
14                  None => {
15                      f(&to_iter.val);
16                      break;
17                  }
18                  Some(ref next) => {
19                      f(&to_iter.val);
20                      // Salva il successore e
21                      // prosegui con l'iterazione.
22                      cp = Rc::clone(next);
23                  }
24              }
25              drop(to_iter);
26              curr = cp;
27          }
28      }
29  }
```

In questo caso ci limitiamo a scorrere la lista e per ogni nodo attraversato invochiamo la chiusura passata come parametro sul valore del nodo. Anche in questo caso dobbiamo clonare il reference counter del successore per l'iterazione successiva, perché il riferimento al nodo precedente ottenuto tramite il metodo *borrow* di RefCell viene invalidato alla fine di ogni iterazione.

Liste multi-thread, usando Arc e Mutex

Una possibile definizione di liste concatenate doppie per l'utilizzo multi-thread è la seguente:

```
1  struct SafeConcurrentNode<T> {
2      val: T,
3      next: Option<Arc<Mutex<SafeConcurrentNode<T>>>>,
4      prev: Option<Weak<Mutex<SafeConcurrentNode<T>>>>,
5  }
6
7
8  impl<T> SafeConcurrentNode<T> {
9      // Metodo costruttore per la lista (testa della lista).
10     pub fn new(item: T) -> Arc<Mutex<SafeConcurrentNode<T>>> {
11         Arc::new(Mutex::new(SafeConcurrentNode {
12             val: item,
13             prev: None,
14             next: None,
15         }))
16     }
17 }
```

La definizione è analoga al caso single thread. Anche in questo caso abbiamo bisogno di un reference counter per violare la regola del singolo owner per un nodo e una cella che ci offre interior mutability per accedere in scrittura e lettura attraverso riferimenti immutabili. A differenza del caso single-thread il reference counter deve avere un meccanismo di sincronizzazione per gestire le modifiche al contatore interno, quindi non possiamo usare Rc e dobbiamo usare Arc (atomic reference counter). Inoltre, anche la cella che offre interior mutability ha bisogno di un meccanismo di sincronizzazione per evitare che più threads possano accedere contemporaneamente al nodo tramite borrow che portano alla violazione della regola AXM, quindi in questo caso possiamo usare Mutex o RwLock (in questo caso scegliamo Mutex).

Il metodo per aggiungere un nodo con valore 'v' dopo la posizione idx-esima posizione è analogo alla versione single-thread e può essere implementato nel modo seguente:

```rust
impl<T> SafeConcurrentNode<T> {
    // head: atomic reference counter alla testa della lista.
    // v: valore da aggiungere.
    // idx: posizione desiderata dove inserire come successore.
    pub fn add_next(head: Arc<Mutex<SafeConcurrentNode<T>>>,
        v: T, idx: u32) {
        // Corpo del metodo analogo al caso con Rc
        let mut curr = head;
        let mut cp;
        let mut index = 0;

        // Scorri la lista fino al nodo desiderato.
        while index < idx {
            match curr.lock().unwrap().next {
                None => {
                    break;
                }
                Some(ref next) => {
                    cp = Arc:.clone(next);
                    index += 1;
                }
            }
            curr = cp;
        }

        // Crea un nuovo nodo da inserire.
        let val = SafeConcurrentNode::new(v);
        val.lock().unwrap().prev = Some(Arc::downgrade(&curr));

        // Aggiorna opportunamente i successori e i predecessori
        let mut curr_locked = curr.lock().unwrap();
        match curr_locked.next {
```

```
33              None => {}
34              Some(ref node) => {
35                  val.lock().unwrap().next = Some(Arc::clone(node));
36                  node.lock().unwrap().prev =
37                      Some(Arc::downgrade(&val));
38              }
39          }
40
41          curr_locked.next = Some(Arc::clone(&val));
42      }
43  }
```

L'unica differenza rispetto all'ambito single-thread è, che per ogni nodo che vogliamo visitare dobbiamo acquisire la lock tramite il Mutex. Precisiamo che le lock una volta acquisite vengono rilasciate in automatico alla fine dello scope, oppure vengono immediatamente rilasciate dopo la fine di un comando se il valore restituito dal metodo *lock* non viene assegnato a nessuna variabile.

Anche il metodo per rimuovere un elemento nella posizione idx-esima è analogo alla versione single-thread e può essere implementato nel modo seguente:

```
1  impl<T> SafeConcurrentNode<T> {
2      // head: atomic reference counter alla testa della lista.
3      // idx: posizione da rimuovere. Il metodo non gestisce
4      // il caso limite di rimozione della testa, quindi
5      // l'indice passato deve essere maggiore di 0.
6      pub fn remove_at(head: Arc<Mutex<SafeConcurrentNode<T>>>,
7          idx: u32) -> Option<T> {
8          // Corpo del metodo analogo al caso con Rc
9          let mut curr = head;
10         let mut cp;
11         let mut index = 0;
12
13         // Scorri fino al nodo desiderato.
```

```
14          while index < idx {
15              match curr.lock().unwrap().next {
16                  None => {
17                      break;
18                  }
19                  Some(ref next) => {
20                      cp = Arc::clone(next);
21                      index+=1;
22                  }
23              }
24              curr = cp;
25          }
26
27          // Nel caso di indice 0 non fare niente (testa della lista).
28          if index == 0 {
29              return None;
30          }
31
32          // Salva il nodo da rimuovere.
33          let to_remove = curr.lock().unwrap();
34
35          // Aggiorna i campi del nodo predecessore.
36          match to_remove.prev {
37              None => {}
38              Some(ref node) => {
39                  match to_remove.next {
40                      None => {
41                          node.upgrade().unwrap().lock().unwrap().next =
42                              None;
43                      }
44                      Some(ref next) => {
45                          node.upgrade().unwrap().lock().unwrap().next =
```

```
46                              Some(Arc::clone(next));
47                      }
48                  }
49              }
50          }
51
52          // Aggiorna i campi dal nodo successore.
53          match to_remove.next {
54              None => {}
55              Some(ref node) => {
56                  match to_remove.prev {
57                      None => {
58                          node.lock().unwrap().prev = None;
59                      }
60                      Some(ref prev) => {
61                          node.lock().unwrap().prev
62                              = Some(Arc::clone(prev));
63                      }
64                  }
65              }
66          }
67
68          // Rilascia il reference counter in modo da
69          // avere solo un riferimento strong attivo.
70          drop(to_remove);
71          Some(Arc::try_unwrap(curr).ok().unwrap().
72              into_inner().unwrap().val)
73      }
74  }
```

Il metodo anche in questo caso necessita di acquisire le lock sui nodi sui quali deve operare e le rilascia immediatamente dopo il comando se non sono assegnate a nessuna variabile per evitare deadlocks.

Anche il metodo iter segue la logica dell'ambito single thread e può essere implementato nel seguente modo:

```
1       // head: atomic reference counter alla testa della lista.
2       // f: chiusura da invocare su ogni nodo durante l'iterazione.
3       pub fn iter(head: Arc<Mutex<SafeConcurrentNode<T>>>,
4           mut f: impl FnMut(&T)) {
5           // Corpo del metodo analogo al caso con Rc
6           let mut curr = head;
7           let mut cp;
8           loop {
9               let to_iter = curr.lock().unwrap();
10              match to_iter.next {
11                  None => {
12                      f(&to_iter.val);
13                      break;
14                  }
15                  Some(ref next) => {
16                      f(&to_iter.val);
17                      cp = Arc::clone(next);
18                  }
19              }
20              drop(to_iter);
21              curr = cp;
22          }
23      }
24  }
```

Ogni volta che si vuole accedere a un nodo si acquisisce la lock e non la liberiamo fino a che l'esecuzione della funzione di iterazione su quel nodo non è terminata.

Con questa definizione a differenza di prima dobbiamo fare attenzione quando utilizziamo gli ac-

cessi in modifica, perché questa struttura non offre una protezione dai deadlocks e quindi se più threads eseguono delle modifiche, pur essendo garantita la memory safety, si potrebbero comunque verificare per via degli aggiornamenti sui nodi adiacenti al nodo modificato, che richiedono l'acquisizione della lock sull'interno nodo per effettuare modifiche ai campi predecessore e successore.
Dopo aver analizzato i metodi del caso multi-thread e dopo aver notato che sono molto simili a quelli del caso single-thread, potremmo pensare che questa definizione non ha senso e potremmo invece racchiudere la lista dell'ambito single-thread dentro un Mutex per renderla thread-safe nel modo seguente:

```
1  struct InvalidList<T> {
2      list: Mutex<SafeNode<T>>
3  }
```

In realtà non si può fare, perché Mutex è per un utilizzo generico e fornisce sincronizzazione per l'accesso al valore contenuto, ma non rende thread safe valori che non lo sono (nel nostro caso la struttura SafeNode). Se infatti andiamo a leggere i requisiti necessari affinché Mutex implementi Send e Sync per l'utilizzo multi-thread sulla documentazione, noteremo che il tipo contenuto deve soddisfare alcuni requisiti:

```
1  // Attenzione! La firma è esattamente uguale!
2  // Il vincolo ?Sized sul generic è stato rimosso
3  // non essendo di nostro interesse per la spiegazione
4  impl<T: Send> Send for Mutex<T>
5  impl<T: Send> Sync for Mutex<T>
```

Il tipo contenuto a sua volta deve implementare il trait Send come imposto dal limite sul generic T, affinché il Mutex lo implementi. Nel nostro caso, la struttura SafeNode non implementa il trait Send, perché al suo interno sono presenti dei campi (Rc) che non implementano Send, quindi la struttura automaticamente non implementa Send e senza utilizzare codice unsafe per forzare l'implementazione, non è possibile inserirla dentro un Mutex o RwLock per essere condivisa tra thread diversi (rimane comunque possibile inserirla ma solo per l'utilizzo single-thread). L'unico modo per risolvere questo problema è utilizzare strutture a sua volta thread-safe per ogni nodo e questo implica che ogni nodo deve usare Arc e (Mutex, RwLock o simili). Ovviamente anche in

questo caso è comunque richiesto che il tipo del valore inserito nella lista (T) deve a sua volta implementare Send, affinché la struttura ConcurrentSafeNode implementi Send (come richiesto da Mutex) e sia condivisibile tra thread diversi.

Liste con GhostCell

Di seguito analizzeremo solo un caso di definizione anziché uno per l'ambito single-thread e uno per l'ambito multi-thread, perché GhostCell ha il vantaggio di rendere tutti i tipi ricorsivi implementati con esso automaticamente thread safe. Questa cosa è possibile perché si sfrutta il meccanismo di ownership e borrowing fornito da Rust sul token per regolare gli accessi alla struttura ricorsiva. Infatti per poter accedere in modo mutabile alla struttura è necessario che il thread sia l'owner del token per poter creare un riferimento mutabile, mentre per accedere in modo immutabile è sufficiente che il token sia condiviso tra i vari threads tramite riferimento immutabile.

Le liste concatenate doppie con GhostCell possono essere definite nel seguente modo:

```
1  struct GhostNode<'arena, 'id, T> {
2      val: T,
3      prev: Option<NodeRef<'arena, 'id, T>>,
4      next: Option<NodeRef<'arena, 'id, T>>
5  }
6  // NodeRef è un riferimento a una GhostCell che contiene un nodo
7  type NodeRef<'arena, 'id, T> =
8      &'arena GhostCell<'id, GhostNode<'arena, 'id, T>>;
9  impl<'arena, 'id, T> GhostNode<'arena, 'id, T> {
10     // Metodo costruttore per la lista (testa della lista).
11     pub fn new(arena: &'arena Arena<GhostCell<'id,
12         GhostNode<'arena, 'id, T>>>, v: T)
13         -> NodeRef<'arena, 'id, T> {
14         arena.alloc(GhostCell::new(GhostNode {
15             val: v,
16             prev: None,
17             next: None,
```

```
18          }))
19      }
20  }
```

In questo caso il predecessore e il successore dei nodi come possiamo notare non è un reference counter a un altro nodo, infatti è semplicemente un riferimento immutabile a una GhostCell che contiene un nodo che ha un lifetime uguale a tutti gli altri nodi della struttura **'arena** e ha lo stesso brand **'id**. L'allocazione dei nodi non può essere fatta normalmente, infatti affinché tutti i riferimenti ai nodi abbiano lo stesso lifetime dobbiamo utilizzare un arena, che per definizione, permette di allocare nodi singolarmente, ma la deallocazione dei nodi viene fatta in blocco quando la arena viene distrutta e non permette la deallocazione dei singoli nodi in modo che tutti i nodi abbiano esattamente lo stesso lifetime e non ci siano problemi durante le modifiche dei riferimenti ai nodi successori e predecessori per l'eventuale possibilità che il lifetime di un nodo possa avere una validità inferiore a quella richiesta, rifiutando di conseguenza l'inserimento.

Il metodo per aggiungere un nodo con valore 'v' dopo la idx-esima posizione può essere implementato passando la arena su cui allocare (che deve avere un lifetime uguale a quello della lista di nodi su cui stiamo chiedendo l'inserimento) nel modo seguente:

```
1  impl<'arena, 'id, T> GhostNode<'arena, 'id, T> {
2      // head: riferimento alla testa della lista.
3      // arena: l'arena dove allocare i nodi della lista.
4      // v: valore da inserire.
5      // idx: posizione desiderata dove inserire come successore.
6      // token: riferimento mutabile al token associato alla lista.
7      pub fn add_next(head: NodeRef<'arena, 'id, T>,
8          arena: &'arena Arena<GhostCell<'id,
9              GhostNode<'arena, 'id, T>>>,
10         v: T, idx: u32, token: &mut GhostToken<'id>) {
11         // Il corpo del metodo rimane simile al caso
12         // con i reference counter, solo che in questo
13         // caso operiamo usando esclusivamente riferimenti.
14         let mut curr = head;
```

```
15          let mut index = 0;
16
17          // Scorri la lista fino al nodo desiderato usando i riferimenti.
18          while index < idx {
19              match curr.borrow(token).next {
20                  None => {
21                      break;
22                  }
23                  Some(node) => {
24                      curr = node;
25                      index+=1;
26                  }
27              }
28          }
29
30          // Crea un nuovo nodo all'interno dell'arena
31          let val = GhostNode::new(arena, v);
32          val.borrow_mut(token).prev = Some(curr);
33
34          // Aggiorna i riferimenti nei nodi adiacenti opportunamente.
35          match curr.borrow(token).next {
36              None => {}
37              Some(node) => {
38                  val.borrow_mut(token).next = Some(node);
39                  node.borrow_mut(token).prev = Some(val);
40              }
41          }
42
43          curr.borrow_mut(token).next = Some(val);
44      }
45  }
```

Il metodo prende in ingresso un riferimento mutabile al token associato alla struttura per poter ac-

cedere in modo mutabile ai nodi tramite i metodi offerti da Ghostcell che permettono di garantire il rispetto delle regole anche in presenza di condivisione tramite riferimenti immutabili. Il metodo scorre la lista sfruttando i metodi di GhostCell per accedere ai nodi e una volta trovata la posizione dove aggiungere il nodo all'interno della lista, il metodo richiede un'allocazione di una nuova GhostCell che contiene il valore all'interno dell'arena e esegue un borrow mutabile sui nodi interessati tramite il passaggio del token e modifica opportunamente il riferimento ai predecessori e ai successori del nodo attuale e dei nodi adiacenti opportunamente.

Il metodo per rimuovere il nodo alla posizione idx-esima all'interno della lista è molto simile al metodo per aggiungere e può essere implementato nel modo seguente:

```
1       // head: riferimento alla testa della lista.
2       // idx: posizione da rimuovere. Il metodo non gestisce
3       // il caso limite di rimozione della testa, quindi
4       // l'indice passato deve essere maggiore di 0.
5       // token: riferimento mutabile al token associato alla lista.
6       pub fn remove_at(head: NodeRef<'arena, 'id, T>,
7           idx: u32, token: &mut GhostToken<'id>) {
8           // Il corpo del metodo rimane simile al caso
9           // con i reference counter, solo che in questo
10          // caso operiamo usando esclusivamente riferimenti
11          let mut curr = head;
12          let mut index = 0;
13
14          // Scorri la lista fino al nodo desiderato.
15          while index < idx {
16              match curr.borrow(token).next {
17                  None => {
18                      break;
19                  }
20                  Some(node) => {
21                      curr = node;
```

```
22                      index += 1;
23                  }
24              }
25          }
26
27          // Nel caso di indice 0 non fare niente (testa della lista).
28          if index == 0 {
29              return;
30          }
31
32          // Aggiorna il nodo successore
33          match curr.borrow(token).next {
34              None => {}
35              Some(node) => {
36                  let next = curr.borrow(token).next;
37                  node.borrow_mut(token).next = next;
38              }
39          }
40
41          // Aggiorna il nodo predecessore
42          match curr.borrow(token).succ {
43              None => {}
44              Some(node) => {
45                  let prev = curr.borrow(token).prev;
46                  node.borrow_mut(token).prev = prev;
47              }
48          }
49      }
```

Anche in questo caso scorriamo la lista fino a trovare il nodo che ci interessa accedendo ai riferimenti dei successori eseguendo un borrow immutabile sul riferimento attuale passando il token in versione immutabile. Una volta trovato il riferimento al nodo che ci interessa, sfruttiamo ancora

una volta i metodi offerti da GhostCell per accedere in modo mutabile ai nodi interessati e modifichiamo opportunamente i campi che contengono i riferimenti ai nodi successori e predecessori.

Il metodo per eseguire iterazioni in modo immutabile sulla lista invece può essere implementato nel modo seguente:

```
1       // head: riferimento alla testa della lista.
2       // token: riferimento immutabile al token associato alla lista.
3       // f: chiusura da invocare su ogni nodo durante l'iterazione.
4       pub fn iter(head: NodeRef<'arena, 'id, T>,
5           token: &GhostToken<'id>, mut f: impl FnMut(&T)) {
6           // Il corpo del metodo rimane simile al caso
7           // con i reference counter, solo che in questo
8           // caso operiamo usando esclusivamente riferimenti
9
10          let mut curr = Some(head);
11          loop {
12              match curr {
13                  None => {
14                      break;
15                  }
16                      Some(node) => {
17                          f(&node.borrow(token).val);
18                          curr = node.borrow(token).succ;
19                      }
20              }
21          }
22      }
23  }
```

In questo caso, visto che non accediamo in modifica ai nodi, richiediamo in ingresso un riferimento immutabile al token, in modo che sia possibile invocare solo il metodo *borrow* offerto da GhostCell

per accedere in sola lettura ai nodi della lista e quindi accettare che il token venga condiviso tra più utilizzatori.

4.3 Analisi dello smart pointer

Confronto

Utilizzare gli smart pointers classici introduce molti costi aggiuntivi dovuti alla necessità di violare le regole imposte da compilatore eseguendo controlli a tempo di esecuzione. Ad esempio per il caso single-thread abbiamo il contatore interno per Rc e altri contatori interni per RefCell per garantire il rispetto della regola AXM, mentre per il caso multi-thread abbiamo sia contatori interni, sia il meccanismo delle lock per evitare race conditions. Utilizzare GhostCell, invece non richiede alcun costo aggiuntivo, perché ci limitiamo a utilizzare semplicemente dei riferimenti immutabili a delle GhostCell come campi della struttura ricorsiva e a sua volta le GhostCell non introducono alcun costo aggiuntivo. La memory safety è garantita in entrambi i casi, ma a differenza del primo caso dove la safety deriva da controlli a tempo di esecuzione, GhostCell la garantisce spostando la verifica del meccanismo di ownership e borrowing per controllare gli accessi sul token anziché sulla struttura ricorsiva. Inoltre, per via di questo controllo degli accessi utilizzando il meccanismo di ownership e borrowing di base, automaticamente tutte le strutture definite con GhostCell hanno il vantaggio di essere thread-safe.

Vantaggi e Svantaggi

L'utilizzo dello smart pointer GhostCell introduce dei vantaggi, ma introduce anche alcuni svantaggi.

Vantaggi:

- Non utilizza operazioni aggiuntive a tempo di esecuzione per accedere in lettura e scrittura alla struttura ricorsiva.
- Offre la possibilità di definire qualsiasi tipo ricorsivo anche contenente cicli utilizzando solamente codice safe tramite riferimenti immutabili.

- Permette di implementare qualsiasi tipo ricorsivo che può essere utilizzato anche in ambito multi-thread garantendo la memory safety con un'unica definizione.
- Garantisce l'assenza di deadlocks nel caso multi-thread, visto che non sono necessarie le lock per definire tipi ricorsivi con questo smart pointer.

Svantaggi:

- L'utilizzo delle arene non permette di deallocare singolarmente i nodi della struttura ricorsiva, bensì tutti i nodi allocati nell'arena vengono deallocati in blocco nel momento in cui termina lo scope nella quale è definita. Questa cosa, a seconda degli utilizzi, può causare leak di memoria anche molto grandi.
- L'utilizzo di un unico token per l'accesso mutabile alla struttura impone la restrizione di avere al più un nodo della struttura ricorsiva con accesso mutabile in ogni istante di esecuzione per evitare di violare la regola AXM di Rust nel caso di strutture cicliche.
- L'utilizzo forzato del meccanismo dei lifetimes in un contesto dove non dovrebbero esserci quando i vincoli del compilatore sono già molto forti, impone dei limiti all'espressività del codice e quindi rendere poco flessibili le implementazioni delle strutture ricorsive (Ad esempio il programmatore è costretto a usare sempre e solo le arene o strutture analoghe dove i nodi non possono essere deallocati singolarmente).

Possibili ottimizzazioni

Normalmente per via dell'utilizzo delle arene potremmo avere dei leak di memoria non indifferenti se la struttura ricorsiva rimane attiva per tanto tempo, però in alcuni casi è comunque possibile sfruttare una piccola ottimizzazione che consiste nel definire più arene con profondità di scope differenti, per cercare di bilanciare il numero di nodi allocati in ciascuna arena e rilasciare periodicamente la memoria delle arene più interne come se fossero organizzate in generazioni. Questo approccio ha comunque un limite, cioè non funziona nel caso in cui un nodo definito in una arena più esterna viene impostato come successore o predecessore di un nodo definito in un arena più interna. Per spiegare questo limite, proveremo a implementare le liste concatenate singole cercando di organizzare l'allocazione dei nodi in più arene.

```
struct GhostNode<'a, 'b, T> {
    val: T,
    next: Option<&'a GhostCell<'b, GhostNode<'a, 'b, T>>>
}
```

Se utilizziamo le arene annidate provando a collegare un nodo di un'arena più esterna verso un nodo di un'arena più interna (esterno → interno) e poi continuiamo a usare la arena più esterna dopo che la memoria dell'arena più interna viene liberata:

```
// Da questo esempio possiamo vedere anche come si usa e possiamo
// notare che l'utilizzo è fatto all'interno della chiusura passata
// al metodo new di GhostToken e l'allocazione è fatta separatamente
fn main() {
    GhostToken::new(|token| {
        let arena1 = Arena::new();
        let cell1;
        {
            let arena2 = Arena::new();
            let cell2 = arena2.alloc(
                GhostCell::new(
                    GhostNode {
                        val: 1,
                        next: None
                    }
                ));

            cell1 = arena1.alloc(
                GhostCell::new(
                    GhostNode {
                        val: 2,
                        next: Some(cell2)
                    }
                ));
```

```
25              let c2 = cell1.borrow(&token);
26              let next = c2.next.unwrap();
27              println!("V1: {}, V2: {}",
28                  next.borrow(&token).val, c2.val);
29          }
30
31          arena1.alloc(GhostCell::new(
32              GhostNode {
33                  val: 1,
34                  next: None
35              }
36          ));
37      });
38  }
```

otteniamo un errore di questo tipo:

```
error[E0597]: `arena2` does not live long enough
   --> src\main.rs:278:25
    |
278 |               let cell2 = arena2.alloc(
    |  _________________________^
279 | |                 GhostCell::new(
280 | |                     GhostNode {val: 1, succ: None }
281 | |                 ));
    | |_________________^ borrowed value does not live long enough
...
289 |           }
    |           - `arena2` dropped here while still borrowed
290 |
291 | /         arena1.alloc(GhostCell::new(
292 | |             GhostNode {val: 1, succ: None }
293 | |         ));
    | |_________- borrow later used here
```

Questo succede, perché un nodo della arena più esterna è collegato a un nodo dell'arena più interna. Se il codice si fermasse dopo lo scope più interno non ci sarebbero problemi, ma siccome stiamo continuando a usare la arena più esterna, otteniamo un errore dovuto alla diminuzione del lifetime a quello della arena più interna per poter permettere la creazione dei collegamenti verso i nodi con

lifetime minore.

Una soluzione possibile per implementare questa ottimizzazione con GhostCell è in questo modo:

```
fn main() {
    GhostToken::new(|token| {
        let arena1 = Arena::new();
        let cell1;
        {
            let arena2 = Arena::new();
            cell1 = arena1.alloc(
                GhostCell::new(
                    GhostNode {
                        val: 2,
                        next: None
                    }
                ));
            let cell2 = arena2.alloc(
                GhostCell::new(
                    GhostNode {
                        val: 1,
                        next: Some(cell1)
                    }
                ));
            let c2 = cell2.borrow(&token);
            let next = c2.next.unwrap();
            println!("V1: {}, V2: {}",
                next.borrow(&token).val, c2.val);
        }

        arena1.alloc(GhostCell::new(
            GhostNode {
                val: 1,
```

```
30                  next: None
31              }
32          ));
33      });
34  }
```

In questo caso il compilatore non genera nessun errore, perché abbiamo messo in comunicazione un nodo dell'arena più interna verso il nodo dell'arena più esterna (interno $\rightarrow$ esterno) e non sono presenti collegamenti da nodi esterni verso nodi interni (esterno $\rightarrow$ interno). Notiamo quindi da questo esempio, che GhostCell per definire una API generica per definire qualsiasi tipo ricorsivo in ambito single-thread e multi-thread, non permette molte ottimizzazioni e il suo utilizzo può portare alla creazione di leak di memoria anche molto grandi a seconda degli utilizzi.

Benchmarks

In questo paragrafo presentiamo dei benchmarks sulle liste implementate nei paragrafi precedenti utilizzando gli stessi metodi presentati e discussi per aggiungere, rimuovere e iterare. In particolare eseguiamo i benchmarks analizzando i casi pessimi, cioè aggiunta e rimozione di elementi al centro della lista e iterazioni sia in ambito single-thread, sia in ambito multi-thread sull'intera lista. Precisiamo che per centro della lista intendiamo il valore $round(list.length()/2)$, cioè il centro della lista attuale o rimanente. I benchmarks sono stati eseguiti utilizzando la libreria criterion [5] per effettuare benchmark per analizzare il tempo impiegato da un frammento di codice su Rust e sono eseguiti con i seguenti criteri:

- Per tutti i casi di benchmark sono state effettuate 100 prove (iterazioni) e come risultato finale è stata presa la media di tutte le prove.
- Per il benchmark sull'inserimento al centro della lista, è stato generato un campione di 10000 interi da inserire, iniziando ogni volta con una lista vuota e inserendo un elemento per volta. Questo test è stato eseguito avendo cura di non far terminare lo scope di definizione della lista che provocherebbe anche il rilascio della memoria (rimozione di tutti i nodi).
- Per il benchmark sulla rimozione al centro, è stata usata una lista già piena, contenente esattamente 10000 elementi e rimuovendo un elemento per volta.

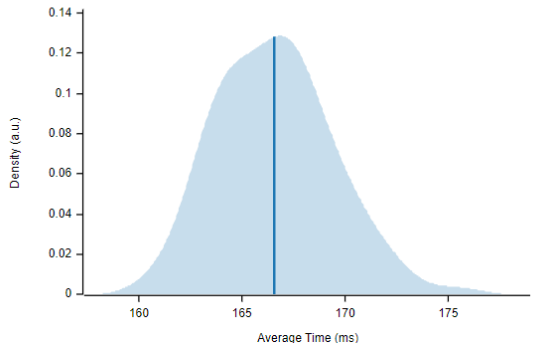
- Per il benchmark sull'iterazione è stata utilizzata una lista contenente esattamente 10000 elementi, eseguendo la somma dei valori dei nodi come funzione di iterazione. Inoltre per il caso multi-thread è stata utilizzata la libreria rayon [19] per eseguire computazioni in parallelo utilizzando il parallelismo di tipo fork-join.

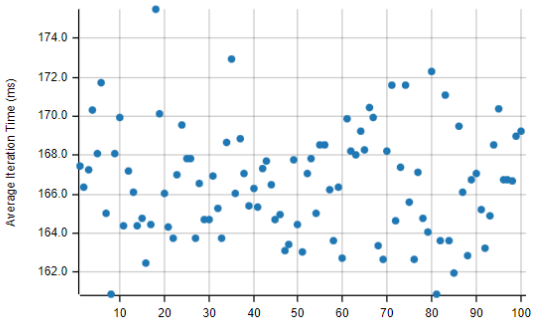
	Inserimento di 10000 valori - Centro lista	Rimozione di 10000 valori - Centro lista	Iterazione con 10000 valori - 1 Thread	Iterazione con 10000 valori - 4 Threads
Liste concatenate doppie - Rc, RefCell	166.49 ms	49.91 µs	65.44 µs	Non compatibile
Liste concatenate doppie - Arc, Mutex	588.34 ms	283.84 ms	313.40 µs	1.13 ms
Liste concatenate doppie - GhostCell	202.09 µs	75.87 µs	11.81 µs	54.85 µs

Notiamo che le prestazioni tra le varie versioni di lista sono molto diverse tra loro, questo accade a causa della presenza o meno di contatori interni e utilizzo o meno delle lock per regolare gli accessi. GhostCell per via della totale assenza di costi aggiuntivi, ha delle prestazioni molto superiori sia alla versione con Rc per il solo uso single-thread, sia alla versione con Arc per l'uso anche multi-thread.

A titolo di esempio riportiamo anche due grafici che mostrano rispettivamente la densità del tempo impiegato dalle varie prove (iterazioni di benchmark) e il tempo impiegato da ciascuna prova per l'aggiunta dei nodi al centro della lista nel caso di Rc (Rc add):

Rc add





5. Un nuovo smart pointer: GenerationalGraph

GenerationalGraph (G^2) è un nuovo tipo di smart pointer progettato specificatamente per l'implementazione di grafi in ambito single-thread e multi-thread, che si pone l'obiettivo di creare una API per la definizione di grafi suddivisi in generazioni senza pagare alcun costo aggiuntivo a tempo di esecuzione in Rust utilizzando esclusivamente codice safe. G^2 sfrutta il meccanismo di analisi statica dei lifetimes e il meccanismo di ownership e borrowing in maniera avanzata per validare i puntatori utilizzati, garantire la memory safety e permettere la creazione di archi tra nodi di generazioni diverse in sicurezza. G^2 non è una semplice specializzazione di GhostCell, infatti non lo utilizza e non modifica la sua definizione, bensì prende gli aspetti più interessanti introdotti da quest'ultimo per creare una API migliore specializzata per l'implementazione di grafi. L'implementazione che vedremo, ovviamente non si limita a questa particolare struttura dati e può essere usata come base per implementare altri tipi ricorsivi nello stesso modo (per esempio le liste concatenate oppure gli alberi essendo casi speciali di grafo). Precisiamo che G^2 consente di creare e manipolare grafi pesati, etichettati, ciclici, diretti e semplici.

GhostCell permette una definizione di grafo in questo modo:

```
1  struct Node<'arena, 'id, T> {
2      data : T,
3      edges : Vec<NodeRef<'arena, 'id, T>>
4  }
5  type NodeRef<'arena, 'id, T> =
6          &'arena GhostCell<'id, Node<'arena, 'id, T>>;
```

Dall'analisi effettuata sui vantaggi e gli svantaggi offerti da GhostCell (Capitolo 4) per implementare tipi ricorsivi, abbiamo osservato che per via della sua natura 'generica', non permette né di avere più nodi con accesso mutabile in contemporanea per via dell'utilizzo di un unico token con un unico brand, né di effettuare collegamenti da nodi definiti in arene più esterne verso arene più interne in modo da ottimizzare l'utilizzo della memoria. Il motivo principale che ci spinge alla definizione di G^2 è quello di proporre una implementazione migliore dell'idea di GhostCell per implementare

grafi in Rust utilizzando esclusivamente codice safe (cioè senza usare puntatori) senza aggiungere costi a tempo di esecuzione. In particolare l'obiettivo di G^2 è garantire accesso mutabile a più nodi contemporaneamente e offrire la possibilità di suddividere i nodi in memory allocators diversi in modo da creare una struttura basata su generazioni in modo da ottimizzare l'utilizzo della memoria per via dell'implementazione necessaria per il suo corretto funzionamento. G^2 (come GhostCell) utilizza le arene per gestire l'allocazione dei nodi all'interno dei grafi. Le arene sono un tipo di memory allocator, che permette di allocare valori singolarmente su richiesta, mentre non permette la deallocazione dei valori singoli e la memoria utilizzata da tutti i valori allocati viene rilasciata solo quando le arene vengono distrutte. Questa proprietà consente di avere dei confini espliciti di validità per i nodi allocati in un'arena che sono guidati dalla struttura statica del codice e ciò consente di effettuare controlli sulla validità dei nodi a tempo di compilazione sfruttando l'analizzatore statico offerto da Rust.

5.1 Definizione delle strutture

In questa sezione discuteremo il contenuto delle strutture necessarie per il funzionamento di G^2 e giustificheremo la necessità di alcuni campi in alcune strutture che normalmente potrebbero sembrare inutili.

Lifetimes

```
1  pub struct CovariantLifetime<'a>(PhantomData<&'a ()>);
2  pub struct InvariantLifetime<'a>(PhantomData<*mut &'a ()>);
3  pub struct ContravariantLifetime<'a>(PhantomData<fn(&'a ()) -> ()>);
```

G^2 sfrutta tutti e tre i tipi di lifetimes visti nel capitolo 2 per implementare tutte le funzionalità in modo sicuro con più flessibilità rispetto a GhostCell. Nella loro definizione viene utilizzato il tipo speciale PhantomData per forzare una specifica varianza dei lifetimes che verrà poi utilizzata nelle definizioni delle varie strutture necessarie a G^2 per validare i puntatori utilizzati. La soluzione proposta ha il vantaggio di utilizzare i lifetimes senza occupare memoria aggiuntiva a tempo di esecuzione, visto che il compilatore automaticamente cancella ogni occorrenza del tipo speciale PhantomData nelle strutture.

GgToken

```
1  pub struct GgToken<'id> {
2      _brand_marker: InvariantLifetime<'id>,
3  }
```

Per regolare gli accessi a G^2 utilizziamo un token analogo a quello introdotto da Ghostcell. Nel nostro caso, però eviteremo quasi del tutto il suo utilizzo per avere maggiore flessibilità e lo sfrutteremo per dimostrare accesso esclusivo al memory allocator (cioè alla arena) sia in ambito single-thread, sia in ambito multi-thread senza usare le lock come meccanismo di sincronizzazione. Questa struttura ha come campo solamente un marcatore per definire un lifetime invariante.

GenerationalGraph

```
1  pub struct Node<T, G> {
2      links: HashMap<*mut Node<T, G>, G>,
3      value: T
4  }
5
6  pub struct GenerationalGraph<'id, T, G> {
7      nodes: Arena<Node<T, G>>,
8      _brand_marker: InvariantLifetime<'id>,
9  }
```

Per memorizzare l'insieme dei nodi allocati di un grafo utilizziamo la struttura GenerationalGraph che contiene due campi:

- Un campo *nodes*, il cui ruolo è di allocare e fare da container per i nodi del grafo. In questo caso sfruttiamo un'arena dato che permette di allocare nodi su richiesta e gestire in automatico la deallocazione di tutti nodi creati, quando la memoria al grafo viene rilasciata.
- Un campo *_brand_marker*, il cui ruolo è realizzare correttamente l'associazione di un solo token al grafo per regolare gli accessi e dare un brand unico a ogni grafo. Questa associazione non avviene direttamente e come per GhostCell renderemo possibile tutto ciò, mediante alcuni metodi speciali per la creazione di nuove istanze.

I nodi a loro volta sono formati da due campi:

- Un campo *links*, che memorizza gli archi verso altri nodi con relativo peso attraverso una mappa (nodo destinazione - peso arco).
- Un campo *value*, che memorizza l'etichetta del nodo.

NodeRef

```
1  pub struct NodeRef<'a, 'id, 'b, T, G> {
2      ptr: *mut Node<T, G>,
3      _life_marker1: CovariantLifetime<'a>,
4      _brand_marker: InvariantLifetime<'id>,
5      _life_marker2: ContravariantLifetime<'b>
6  }
```

La struttura NodeRef rappresenta il riferimento a un nodo allocato in un'arena di un grafo e permette interagire con il nodo a cui si riferisce tramite metodi speciali, permettendo di alterare l'etichetta del nodo e creare o rimuovere archi verso altri nodi. La struttura è formata da tre campi:

- Un campo *ptr*, che contiene l'indirizzo di memoria per accedere al nodo in modo mutabile.
- Un campo *_life_marker1*, il cui ruolo è legare il lifetime di questa struttura con il lifetime del grafo in modo da invalidare il riferimento (NodeRef) quando la memoria del grafo viene rilasciata (cioè della arena). Da notare che questo lifetime essendo covariante permette di verificare che il nodo destinazione abbia durata maggiore o uguale a quella del nodo di provenienza se questi lifetime vengono messi in relazione tra loro. Ciò consente di creare archi che partono da nodi di generazioni più interne e hanno come destinazione un nodo di un grafo definito più esternamente.
- Un campo *_brand_marker*, il cui ruolo è ereditare il brand associato al grafo per sfruttarlo nelle definizioni dei metodi per la creazione degli archi per controllare che i nodi presi in ingresso (cioè destinazione) appartengano effettivamente allo stesso grafo (controllando che abbiano lo stesso brand). In questo caso è necessario che sia invariante per poter distinguere tra nodi che hanno esattamente la stessa durata oppure no.

- Un campo *_life_marker2*, il cui ruolo è legare il lifetime del grafo in modo contravariante in modo che sia utilizzabile come nodo destinazione dove è richiesto un nodo di una generazione più interna mettendo in relazione 'contravariante' lifetime del nodo di provenienza con il lifetime del nodo di destinazione.

5.2 Implementazione dei metodi

In questa sezione discuteremo una possibile implementazione dei metodi di G^2 adatta per l'utilizzo single-thread e successivamente discuteremo eventuali problematiche relative al multi-thread e relative soluzioni.

GgToken

Questa struttura non definisce alcun metodo, infatti il suo ruolo è quello di regolare l'accesso per l'allocazione di nuovi nodi all'interno di un'arena anche quando il grafo è condiviso tramite riferimento immutabile. Il token viene creato automaticamente insieme al relativo contenitore al momento della creazione di G^2.

GenerationalGraph

La struttura GenerationalGraph definisce vari metodi:

```
1  impl<'id, T, G> GenerationalGraph<'id, T, G> {
2      // f: chiusura a cui viene passata un'istanza di un
3      // nuovo contenitore e un'istanza di un nuovo token.
4      // La chiusura genera anche un lifetime per dare un
5      // brand ai parametri, rendendoli incompatibili con
6      // altri contenitori e token generati da altre invocazioni.
7      pub fn new(f: impl for<'new_brand> FnOnce(
8              GenerationalGraph<'new_brand, T, G>,
9              GgToken<'new_brand>) -> ()
10         ) {
11
12         // Invoca la chiusura con un nuovo contenitore,
```

```
13          // un nuovo token e legali tramite un nuovo lifetime
14          f(GenerationalGraph {
15              nodes: Arena::new(),
16              _brand_marker: InvariantLifetime(PhantomData),
17          },
18            GgToken {
19              _brand_marker: InvariantLifetime(PhantomData),
20            })
21      }
22  }
```

Definisce un metodo *new*, che ha il ruolo di invocare la chiusura passata come parametro con un'istanza del contenitore (GenerationalGraph) e un'istanza del token (GgToken) legandole assieme tramite il lifetime **'new_brand**, in modo da associare un token diverso e un brand diverso per ogni grafo usando una strategia analoga a quanto fatto da GhostCell. A differenza di GhostCell che separava la creazione delle celle dal token, in questo caso abbiamo unito la definizione per via dell'invarianza dei lifetimes che non permettono di avere un token con lifetime invariante condiviso tra più chiusure con diverso livello di annidamento, dato che questa cosa permetterebbe alle arene che hanno validità di una chiusura creata tramite il metodo *new* di 'scappare' in una chiusura sempre creata tramite il metodo *new* più esterno (cosa vietata per via dell'invarianza del lifetime creato sul grafo e necessaria per associare lifetimes diversi a ogni grafo per avere una corretta definizione). Notiamo che per la dichiarazione del metodo non è stato usato il costrutto **fn new**<**'new_brand**> per creare un lifetime, ma è stato usato il costrutto **impl for**<**'new_brand**> [7, 16], perché (come detto nel capitolo 4) il primo costrutto consente di definire un lifetime in funzione del corpo del metodo, il secondo consente di definire un lifetime in funzione della chiusura presa come parametro e permette di istanziare un nuovo lifetime 'fresco' a ogni invocazione. I due costrutti sono diversi e non possiamo usare il primo, in quanto il lifetime viene istanziato al momento dell'invocazione del metodo e non essendo in relazione con nessun altro lifetime, il brand e la durata del grafo non verrebbero assegnati correttamente e avremmo lifetimes 'liberi'.

Questa struttura definisce anche un metodo per creare i nodi all'interno dell'arena:

```
impl<'id, T, G> GenerationalGraph<'id, T, G> {
    // self: riferimento immutabile al contenitore stesso.
    // val: valore da inserire dentro a un nodo.
    // token: riferimento mutabile al token associato in modo
    // da dimostrare accesso esclusivo alla arena.
    // return: riferimento al nodo creato avente lo stesso
    // lifetime 'id (brand) del grafo e validato con il
    // lifetime 'a per associare una durata pari alla
    // durata del grafo stesso.
    pub fn add<'a>(&'a self, val: T, token: &mut GgToken<'id>)
                -> NodeRef<'a, 'id, 'a, T, G> {

        unsafe {
            let node = self.nodes.alloc(
                    Node {
                        value: val,
                        links: HashMap::new()
                    });

            NodeRef {
                ptr: node as *mut Node<T, G>,
                _life_marker1:
                    CovariantLifetime(PhantomData),
                _brand_marker:
                    InvariantLifetime(PhantomData),
                _life_marker2:
                    ContravariantLifetime(PhantomData)
            }
        }
    }
}
```

Il metodo *add*, permette di allocare un nuovo nodo nel grafo. Il metodo richiede un riferimento

immutabile al grafo (self) e un riferimento mutabile al token associato al grafo. Il motivo per cui è stato scelto di separare esplicitamente i permessi dai dati nasce dal fatto che con l'utilizzo di un riferimento immutabile non abbiamo limiti sul numero di riferimenti ai nodi ritornati che possiamo mettere in relazione con il lifetime del grafo. Se usassimo un riferimento mutabile al grafo potremmo validare solo un nodo, poiché il riferimento mutabile al grafo non verrebbe disattivato fino a che il riferimento al nodo ritornato è utilizzato. Il lifetime **'a** è utilizzato sia in posizione covariante, sia in posizione contravariante nel riferimento ritornato dal metodo (NodeRef) per associare la durata del grafo con entrambe le varianze, in modo da poterle sfruttare nei metodi di creazione degli archi che vedremo tra poco. Il lifetime **'id** viene ereditato direttamente dal grafo e viene associato al lifetime invariante del riferimento, in modo da dare un brand anche al riferimento e permettere la creazione di archi tra nodi appartenenti allo stesso grafo senza controlli.
Notiamo che nella dichiarazione del metodo è stato dichiarato un altro generic lifetime **'a** per collegare il lifetime del grafo (self) a quello del riferimento al nodo ritornato. Questo aspetto è essenziale e non è possibile utilizzare il generic lifetime **'id** associato al grafo per effettuare questa associazione, perché altrimenti il riferimento *self* avrebbe durata uguale al corpo della chiusura e il compilatore non riuscirebbe a disattivare il riferimento quando viene richiesto del rilascio della memoria del contenitore al termine del corpo della chiusura. L'idea intuitiva è quella di creare un ciclo di dipendenze, affinché la memoria del grafo sia liberabile, tutti i suoi riferimenti devono essere disattivati (come imposto dal meccanismo di ownership e borrowing), ma poiché i riferimenti a *self* hanno una durata uguale al corpo della chiusura essi non possono essere mai disattivati prima della fine della chiusura.

NodeRef

La struttura NodeRef rappresenta un riferimento a un nodo allocato nell'arena e implementa alcuni trait per smart pointers per offrire maggiore flessibilità di utilizzo:

```
1  impl<'a, 'id, 'b, T, G>
2          Deref for NodeRef<'a, 'id, 'b, T, G> {
3      type Target = T;
4
5      // self: riferimento immutabile al nodo stesso.
```

```
6       // return: un riferimento immutabile al valore del nodo.
7       fn deref<'k>(&'k self) -> &'k T {
8           unsafe { &(*self.ptr).value }
9       }
10  }
11
12  impl<'a, 'id, 'b, T, G>
13          DerefMut for NodeRef<'a, 'id, 'b, T, G> {
14
15      // self: riferimento mutabile al nodo stesso.
16      // return: un riferimento mutabile al valore del nodo.
17      fn deref_mut<'k>(&'k mut self) -> &'k mut T {
18          unsafe { &mut (*self.ptr).value }
19      }
20  }
```

La struttura implementa i trait *Deref* e *DerefMut* per accedere in modo immutabile o mutabile al valore di un nodo. Al contrario di GhostCell, in questo caso non è richiesto il token, perché l'accesso al valore del nodo è regolato dal meccanismo di ownership e borrowing sulla struttura stessa. Per poter creare un riferimento al valore contenuto nel nodo è necessario dereferenziare il puntatore, quindi è necessario dichiarare un blocco unsafe e successivamente estrarre un riferimento al valore contenuto nel nodo. Inoltre, in queste implementazioni è stato dichiarato un generic lifetime **'k** per collegare i lifetime dei riferimenti ricevuti in ingresso e in uscita, in modo da garantire il rispetto della regola AXM, vietando la creazione di altri riferimenti all'etichetta non compatibili con quelli attivi.

Questa struttura definisce anche dei metodi per la gestione degli archi tra nodi intra-grafo:

```
1   impl<'a, 'id, 'b, T, G> NodeRef<'a, 'id, 'b, T, G> {
2
3       // self: riferimento mutabile al nodo di provenienza.
4       pub fn link_self(&mut self, cost: G) {
5           unsafe {
```

```
6               (*self.ptr).links.insert(self.ptr, cost);
7           }
8       }
9
10      // Metodo duale al metodo link_self.
11      pub fn unlink_self(&mut self) {
12          unsafe {
13              (*self.ptr).links.remove(&self.ptr);
14          }
15      }
16
17      // self: riferimento mutabile al nodo di provenienza
18      // other: nodo destinazione (appartenente allo stesso grafo).
19      // Per verificare che appartenga allo stesso grafo utilizziamo
20      // il lifetime invariante 'id che indica il brand del grafo di
21      // appartenenza ed è sempre diverso, grazie alle chiusure
22      // cost: costo dell'arco.
23      pub fn link(&mut self,
24          other: &NodeRef<'a, 'id, '_, T, G>, cost: G) {
25
26          unsafe {
27              (*self.ptr).links.insert(other.ptr, cost);
28          }
29      }
30
31      // Metodo duale al metodo link.
32      pub fn unlink(&mut self, other: &NodeRef<'a, 'id, '_, T, G>) {
33
34          unsafe {
35              (*self.ptr).links.remove(&other.ptr);
36          }
37      }
```

```
38  }
```

Per gestire la creazione e la cancellazione degli archi tra nodi appartenenti allo stesso grafo, la struttura definisce quattro metodi:

- Un metodo *link_self*, che prende in ingresso solo il nodo stesso (self) e crea un arco verso sé stesso con relativo costo (un ciclo).
- Un metodo *unlink_self*, che prende in ingresso solo il nodo di provenienza (self) e rimuovere (se presente) l'arco verso sé stesso.
- Un metodo *link*, che prende in ingresso un riferimento al nodo di provenienza (self), un riferimento a un nodo destinazione appartenente allo stesso grafo con relativo costo e lo inserisce dentro la mappa degli archi del nodo. In questo caso, per essere sicuri che il nodo destinazione appartenga allo stesso grafo, ci interessa che entrambi i nodi abbiano esattamente lo stesso lifetime, quindi possiamo controllare il lifetime invariante **'id** e ignorare i valori degli altri lifetime usando il lifetime anonimo. Il metodo consiste in una dereferenziazione del puntatore per accedere al nodo e inserisce l'arco dentro la mappa, inserendo il puntatore al nodo destinazione insieme al peso dell'arco.
- Un metodo *unlink*, che prende in ingresso un riferimento al nodo di provenienza (self) e un riferimento a un nodo destinazione appartenente allo stesso grafo e lo rimuove (se presente) dalla mappa degli archi del nodo di provenienza. Il corpo del metodo esegue l'operazione duale al metodo link e dopo aver dereferenziato il puntatore al nodo, rimuove dalla mappa l'arco che ha come destinazione il nodo passato come parametro.

Oltre a questi metodi per la gestione di nodi intra-grafo, la stessa struttura definisce anche dei metodi per la gestione dei nodi inter-grafo:

```
1  impl<'a, 'id, 'b, T, G> NodeRef<'a, 'id, 'b, T, G> {
2
3      // Stessi parametri della link, ma a differenza di prima
4      // consideriamo il primo lifetime (covariante) per garantire che la
5      // durata sia maggiore o uguale a quella del nodo di provenienza
6      pub fn link_outer(&mut self,
```

```
7           other: &NodeRef<'a, '_, '_, T, G>, cost: G) {
8           unsafe {
9           (*self.ptr).links.insert(other.ptr, cost);
10          }
11      }
12
13      // Metodo duale al metodo link_outer
14      pub fn unlink_outer(&mut self,
15          other: &NodeRef<'a, '_, '_, T, G>) {
16          unsafe {
17          (*self.ptr).links.remove(&other.ptr);
18          }
19      }
20
21      // Anche in questo caso i parametri sono uguali agli altri casi,
22      // ma consideriamo il terzo lifetime per controllare che abbia
23      // una durata minore o uguale a quella del nodo di provenienza.
24      // Inoltre è richiesta una chiusura per eseguire del codice extra.
25      pub fn link_inner(&mut self, other: &NodeRef<'_, '_, 'a, T, G>,
26          cost: G, with_link: impl FnOnce(
27              &mut NodeRef<'a, 'id, 'b, T, G>) -> ()) {
28          unsafe {
29              (*self.ptr).links.insert(other.ptr, cost);
30              with_link();
31              (*self.ptr).links.remove(&other.ptr);
32          }
33      }
34
35      // Metodo analogo a link_inner per permettere la creazione
36      // di molteplici archi allo stesso tempo. Tutti i nodi
37      // destinazione sono appartenenti allo stesso grafo, perché
38      // la presenza degli altri lifetime rende l'intero vettore
```

```
39      // invariante e non è quindi possibile inserire nodi di altri
40      // grafi avendo lifetime diversi con la nostra implementazione.
41      pub fn link_inners(&mut self,
42          others: Vec<&NodeRef<'_, '_, 'a, T, G>>, costs: Vec<G>,
43          with_link: impl FnOnce(
44              &mut NodeRef<'a,'id,'b, T, G>) -> ()) {
45          unsafe {
46              if others.len() != costs.len() {
47                  return;
48              }
49
50              let mut costs_vec = costs;
51              for index in 0..others.len() {
52              (*self.ptr).links.insert(
53                  others.get_unchecked(index).ptr,
54                  costs_vec.remove(0));
55              }
56              with_link(self);
57              for index in 0..others.len() {
58                  (*self.ptr).links.remove(
59                      &others.get_unchecked(index).ptr
60                  );
61              }
62          }
63      }
64  }
```

Per gestire la creazione e cancellazione degli archi tra nodi di grafi diversi, la struttura definisce quattro metodi:

- Un metodo *link_outer*, che prende in ingresso un riferimento al nodo di provenienza (self), un riferimento a un nodo destinazione appartenente allo stesso grafo o a un grafo più esterno con relativo costo e lo inserisce dentro la mappa degli archi del nodo. Notiamo che per

controllare che il metodo accetti effettivamente solo riferimenti a nodi che hanno un lifetime maggiore o uguale a quello del nodo su cui viene invocato, il metodo controlla che il lifetime covariante del nodo in ingresso abbia un lifetime maggiore rispetto al lifetime (**'a**) sul nodo di provenienza e ignora il valore degli altri lifetime tramite il lifetime anonimo. Il corpo del metodo in questo caso non esegue operazioni aggiuntive rispetto al metodo link e si limita a creare l'arco inserendo il nodo destinazione all'interno della mappa.

- Un metodo *unlink_outer*, che prende in ingresso un riferimento al nodo di provenienza (self) e un riferimento a un nodo destinazione appartenente allo stesso grafo o a un grafo più esterno e lo rimuove dalla mappa degli archi del nodo.

- Un metodo *link_inner*, che prende in ingresso un riferimento al nodo di provenienza (self), un riferimento a un nodo appartenente allo stesso grafo o a un grafo più interno e una chiusura invocabile almeno una volta per eseguire un pezzo di codice con l'arco tra i nodi attivo. Notiamo che per controllare che i riferimenti a nodi abbiano un lifetime minore o uguale, il metodo controlla il valore del lifetime contravariante e ignora gli altri lifetime utilizzando anche in questo caso il lifetime anonimo. Dato che i riferimenti ai nodi passati come parametri non vengono disattivati per tutta la durata del metodo, essi vietano implicitamente il rilascio della memoria dei grafi interessati nell'arco durante l'esecuzione del corpo della chiusura. Per poter continuare a usare il nodo di provenienza dell'arco, passiamo il riferimento mutabile a self alla chiusura stessa, altrimenti non sarebbe più possibile utilizzarlo in quanto non si possono creare altri riferimenti all'interno della chiusura a causa del riferimento mutabile ancora in utilizzo dal metodo. Inoltre, notiamo che dopo aver eseguito la chiusura, per evitare la creazione di dangling references il metodo rimuove il nodo destinazione dalla mappa degli archi.

- Un metodo *link_inners* analogo al metodo precedente, ma consente di creare un numero arbitrario di archi allo stesso tempo.

In tutti i metodi per aggiungere e rimuovere archi, è richiesto un riferimento mutabile al nodo di provenienza per effettuare le modifiche sulla mappa degli archi garantendo il rispetto della regola AXM anziché del token associato al grafo. Questa strategia è stata scelta per migliorare la flessibilità della definizione permettendo di non dipendere dal tipo di riferimento attivo su altri nodi per

accedere in lettura e/o scrittura.

Su questa struttura sono definiti anche alcuni metodi aggiuntivi:

```
impl<'a, 'id, 'b, T, G> NodeRef<'a, 'id, 'b, T, G> {
    pub fn node_id(&self) -> usize {
        self.ptr as usize
    }

    pub fn links_ids(&self) -> Vec<usize> {
        unsafe {
            let mut tmp = Vec::with_capacity((*self.ptr).links.len());
            for (node, cost) in (*self.ptr).links.iter() {
                tmp.push(*node as usize)
            }
            tmp
        }
    }

    pub fn weight_of<'w>(&'w self, dest: usize) -> Option<&'w G> {
        unsafe {
            let ptr = dest as * mut Node<T, G>;
            (*self.ptr).links.get(&ptr)
        }
    }

    pub fn weight_of_mut<'w>(&'w mut self, dest: usize)
        -> Option<&'w mut G> {
        unsafe {
            (*self.ptr).links.get_mut(&(dest as * mut Node<T, G>))
        }
    }
}
```

La struttura definisce quattro metodi per l'eventuale estensione di G^2 attraverso codice safe:

- Un metodo *node_id* per ottenere l'identificatore corrispondente al nodo stesso. Come identificatore, per evitare di occupare memoria aggiuntiva per un campo extra per salvarlo all'interno di un nodo, utilizziamo l'indirizzo di memoria del nodo convertendolo nel tipo usize che può rappresentare numeri senza segno e ha una dimensione di 4 bytes su architetture a 32 bits e 8 bytes su architetture a 64 bits.

- Un metodo *links_uids* per ottenere la lista degli identificatori dei nodi adiacenti (nodi destinazione degli archi). Anche in questo caso gli identificatori sono ottenuti convertendo l'indirizzo di memoria dei nodi destinazione nel tipo usize.

- Un metodo *weight_of* e un metodo *weight_of_mut* per ottenere il peso dell'arco in modo immutabile o mutabile verso il nodo destinazione passato come parametro. Dato che il valore restituito è un riferimento al peso dell'arco, per evitare modifiche al nodo mentre il riferimento è attivo, dobbiamo legare il lifetime del riferimento al peso dell'arco al lifetime del riferimento al nodo stesso (self) passato come parametro. In questo modo il riferimento a self con i relativi permessi di accesso può essere disattivato solo quando il riferimento al peso non è più utilizzato e nel mentre non è possibile accedere al nodo con permessi non coerenti con la regola AXM.

Questi metodi non servono per il normale utilizzo di G^2, ma come vedremo tra poco, permettono di estendere G^2 con i vari algoritmi su grafi utilizzando solo codice safe pagando alcuni costi aggiuntivi.

5.3 Discussione sull'implementazione

Osservazioni sull'implementazione

Notiamo che la soluzione proposta di G^2 è solo una e non sono presenti più versioni per contesti diversi. Inoltre, i metodi di G^2 non impongono una sola struttura generazionale possibile. G^2 infatti consente di creare un numero arbitrario di generazioni con struttura e durata diversa. Precisiamo che questa è solo una delle possibili implementazioni, infatti lo stesso comportamento può essere

ottenuto anche con una dichiarazione diversa. Ad esempio, è possibile rimuovere il lifetime contravariante e accettare tutti i riferimenti ai nodi indipendentemente dal lifetime nel metodo *link_inner* anziché solo quelli che hanno un lifetime minore o uguale. L'utilizzo del lifetime contravariante è stato scelto per maggiore chiarezza e può aiutare a evitare alcuni errori concettuali che si possono creare per via dell'utilizzo del metodo sbagliato nella creazione di un arco. Nel caso in cui un nodo è collegato con un nodo dello stesso grafo, poiché non esiste una varianza di lifetime che permette di accettare in ingresso qualsiasi lifetime **diverso** da quello attuale, è possibile utilizzare i metodi *link_outer*, *link* e *link_inner* indiscriminatamente senza violare la memory safety. Inoltre, osserviamo che l'utilizzo delle chiusure, a differenza di GhostCell che per le varie celle non le utilizza, in questo caso è necessario per definire sia esplicitamente la durata di ogni generazione (che inizia al momento della invocazione della chiusura e termina alla fine del corpo della chiusura stessa), sia per inizializzare correttamente i lifetimes e permettere al compilatore di controllare che non siano presenti degli archi verso grafi per cui la memoria è stata rilasciata.

Algoritmi su grafi

Adesso che abbiamo definito le strutture e i metodi necessari per definire e manipolare grafi suddivisi in generazioni, mostriamo come sia possibile implementare i vari algoritmi che operano su grafi su G^2, analizzando anche eventuali problemi e relative soluzioni. Per implementare gli algoritmi su grafi abbiamo principalmente due possibilità:

- Nei casi in cui è necessario avere la sicurezza di una definizione corretta, possiamo raggruppare i riferimenti ai nodi ritornati dalle allocazioni dentro una struttura di supporto (ad esempio una mappa) e implementare i metodi di un algoritmo utilizzando codice safe, sfruttando l'identificatore univoco del nodo per ricercare i nodi all'interno della struttura. Questa strategia ha come vantaggio il non dover utilizzare codice unsafe per implementare l'algoritmo e come svantaggio i costi in termini di memoria e di tempo per memorizzare e ricercare i nodi all'interno della struttura di supporto.

- Nei casi in cui i costi aggiunti dall'utilizzo di esclusivamente codice safe non sono tollerabili, possiamo utilizzare codice unsafe per estendere la struttura NodeRef per accedere ai puntatori ai nodi in modo da poter implementare l'algoritmo con l'aiuto dei puntatori. Questa strategia ha come svantaggio che chi estende questa dichiarazione deve dimostrare la correttezza della

definizione data e ha come vantaggio l'assenza di costi aggiuntivi a tempo di esecuzione. Vedremo che seguendo alcune regole, la dimostrazione di correttezza può essere semplificata.

Per dare l'idea degli approcci, definiamo la ricerca in ampiezza tramite estensione unsafe e tramite estensione safe.

L'estensione tramite codice unsafe è molto semplice ed è analoga ad implementare l'algoritmo di ricerca in ampiezza su un grafo normale.
Estensione tramite codice unsafe:

```
1  // Codice unsafe, estensione tramite blocco impl e puntatori
2  impl<'a, 'id, 'b, T, G: PartialOrd> NodeRef<'a, 'id, 'b, T, G> {
3      // self: riferimento immutabile al nodo.
4      // init: è richiesto il passaggio di una chiusura in sola lettura
5      // con l'ambiente esterno per creare una struttura di supporto.
6      // each: è richiesto il passaggio di una chiusura in sola lettura
7      // con l'ambiente esterno per eseguire codice personalizzato per
8      // ogni nodo visitato durante l'algoritmo in modo da non
9      // poter avere anche accesso mutabile al nodo tramite un
10     // eventuale cattura nell'ambiente esterno.
11     pub fn bfs<R>(&self,
12         init: impl Fn() -> R,
13         each: impl Fn(&T, &mut R) -> ()) -> R {
14
15         let mut frontier = LinkedList::from([self.ptr]);
16         let mut visited = HashSet::new();
17         let mut supp = init();
18
19         unsafe {
20             // Esegui l'algoritmo bfs
21             while !frontier.is_empty() {
22                 // Prendi il primo nodo in frontiera.
23                 let ptr = frontier.pop_front().unwrap();
```

```
24
25                  // Se il nodo non è stato visitato
26                  if !visited.contains(&ptr) {
27                      // invoca la chiusura sul nodo
28                      each(&(*ptr).value, &mut supp);
29                      // e inserisci il nodo tra quelli visitati.
30                      visited.replace(ptr);
31                      // Metti in frontiera tutti gli archi del nodo.
32                      for (node, weight) in (*ptr).links.iter() {
33                          frontier.push_back(*node)
34                      }
35                  }
36              }
37          }
38          supp // Ritorna la struttura di supporto.
39      }
40  }
41
42  fn main() {
43      GenerationalGraph::new(|graph1, mut token1| {
44          let mut x1 = graph1.add(1, &mut token1);
45          GenerationalGraph::new(|graph2, mut token2| {
46              let mut y1 = graph2.add(2, &mut token2);
47              y1.link_outer(&x1, 2);
48
49              y1.bfs(||{}, |val, supp| {
50                  println!("Node Val={}", val);
51              });
52          });
53      });
54
55      // Stampa: Node Val=2 (nuova linea) Node Val=1.
```

```
56  }
```

L'approccio unsafe consiste nel definire un nuovo metodo sulla struttura NodeRef. Per implementare la ricerca in ampiezza utilizziamo delle strutture di appoggio per tenere la frontiera, la lista dei nodi visitati e una struttura di appoggio fornita dall'utente per eseguire del codice personalizzato e salvare dei valori. Nella versione unsafe, per evitare di violare la regola AXM a causa di un'eventuale cattura di riferimenti a nodi dei grafi, viene richiesta in ingresso una chiusura con accesso in sola lettura all'ambiente catturato in modo che non sia possibile modificare i nodi durante l'iterazione. Precisiamo che non abbiamo ancora parlato di thread safety, quindi questa affermazione per il momento vale solo per l'ambito single-thread.

L'estensione tramite codice safe è diversa dall'estensione unsafe. Con l'utilizzo di codice safe non possiamo usare i puntatori direttamente, quindi i nodi di ogni generazione devono essere inseriti in una mappa e devono essere ricercati tramite un identificatore unico. Il problema principale di questa strategia, però è che non possono essere inseriti in un'unica mappa, poiché hanno lifetime diversi e non compatibili per via dell'invarianza. Per risolvere questo problema dobbiamo usare mappe separate per ogni generazione e a ogni passo dell'algoritmo dobbiamo controllare in ogni lista se è presente il nodo che ci interessa.
Estensione tramite codice safe:

```
1  // Codice safe, estensione tramite mappe.
2  fn visit(frontier: &mut LinkedList<usize>,
3      visited: &mut HashSet<usize>,
4      map: &HashMap<usize, &NodeRef<i32, i32>>,
5      node_id: usize, each: impl FnOnce(&i32) -> ()) {
6
7      // Cerchiamo tramite l'indetificatore se il nodo passato come
8      // parametro è presente nella mappa. Se è presente visita il
9      // nodo e inserisci i suoi archi in frontiera, altrimenti ritorna.
10     match map.get(&node_id) {
11         None => {
12             return;
```

```
13          }
14          Some(node) => {
15              // Controlla che il nodo non sia già stato visitato.
16              if !visited.contains(&node.node_id()) {
17                  // Codice personalizzato.
18                  each(&(***node));
19                  // Aggiungi il nodo insieme a quelli già visitati.
20                  visited.replace(node.node_id());
21                  // Inserisci in frontiera i nodi adiacenti (archi).
22                  for link_id in node.links_ids() {
23                      frontier.push_back(link_id);
24                  }
25              }
26          }
27      }
28  }
29
30  fn main() {
31      GenerationalGraph::new(|graph1, mut token1| {
32          let mut x1 = graph1.add(1, &mut token1);
33          GenerationalGraph::new(|graph2, mut token2| {
34              let mut y1 = graph2.add(2, &mut token2);
35              y1.link_outer(&x1, 2);
36
37              let mut map1 = HashMap::new();
38              map1.insert(x1.node_id(), &x1);
39              let mut map2 = HashMap::new();
40              map2.insert(y1.node_id(), &y1);
41
42              let mut curr = None;
43              let mut frontier = LinkedList::from([y1.node_id()]);
44              let mut visited = HashSet::new();
```

```
45
46              // esegui la bfs.
47              loop {
48                  // Prendi il primo nodo in frontiera.
49                  curr = frontier.pop_front();
50                  match curr {
51                      None => break,
52                      Some(node) => {
53                          // Chiamiamo visit sulla prima mappa.
54                          visit(&mut frontier, &mut visited,
55                              &map1, node, |val| {
56                                  println!("Node Val={}", val);
57                              }
58                          );
59                          // Chiamiamo visit sulla seconda mappa.
60                          visit(&mut frontier, &mut visited,
61                              &map2, node, |val| {
62                                  println!("Node Val={}", val);
63                              }
64                          );
65                      }
66                  }
67              }
68          });
69      });
70
71      // Stampa: Node Val=2 (nuova linea) Node Val=1.
72  }
```

A ogni iterazione estraiamo il prossimo identificatore del nodo dalla frontiera e cerchiamo il nodo interessato chiamando la funzione *visit* su ogni mappa. Se il nodo è presente nella mappa possiamo visitarlo e proseguire inserendo i suoi archi all'interno della frontiera, altrimenti dobbiamo ritornare

e passare alla prossima mappa. L'algoritmo è analogo alla versione unsafe, però non potendo dereferenziare i puntatori direttamente dobbiamo utilizzare una struttura di appoggio che consente la ricerca dei nodi in maniera efficiente.

Thread safety

G^2 è automaticamente thread-safe in quanto gli accessi ai nodi vengono regolati dal normale meccanismo di ownership e borrowing, l'accesso alla arena del grafo è regolato dal token associato (solo un thread per volta può allocare nella arena, perché è richiesto un riferimento mutabile al token) e le chiusure unite al meccanismo dei lifetime permettono di condividere la struttura generazionale tra i vari threads senza permettere il rilascio della parte di struttura in utilizzo. Quest'ultima affermazione non è semplice da notare e potremmo pensare che una struttura di questo tipo non possa essere thread-safe, in quanto si potrebbe rilasciare parte della memoria associata alla struttura ricorsiva, ma questa affermazione non vale per Rust, perché grazie ai lifetimes e al meccanismo di ownership e borrowing applicato sulle chiusure da parte del compilatore, non è possibile rilasciare la memoria di un grafo quando entra in relazione con un altro grafo se questa relazione è utilizzata successivamente (vedremo alcuni esempi nella sezione 5.4).
Il token per accedere all'arena è necessario e non può essere eliminato, perché consultando la documentazione di questo tipo [4], notiamo che implementa i trait per l'utilizzo multi-thread nel modo seguente:

```
1  impl<T: Send> Send for Arena<T> {}
2  impl<T> !Sync for Arena<T> {}
```

Il tipo Arena, implementa Send se e solo se il tipo da gestire lo implementa a sua volta e questo vincolo possiamo soddisfarlo imponendo che G^2 implementi il trait Send se e solo se l'etichetta dei nodi e il peso degli archi lo implementano a loro volta. Notiamo però che questo tipo non implementa Sync in quanto il nome del trait è preceduto dall'operatore not (!), quindi non utilizza meccanismi di sincronizzazione interni per allocare i nodi e dato che è possibile richiedere una nuova allocazione attraverso un semplice riferimento immutabile all'istanza stessa (usando il metodo *alloc*), questo tipo non può essere condiviso tra threads tramite un riferimento immutabile. Nel nostro caso richiedendo il token associato al grafo in versione mutabile come parametro per allocare un nuovo nodo all'interno dell'arena tramite il metodo *add*, automaticamente deleghiamo

al token il compito di regolare gli accessi all'arena secondo il meccanismo di ownership e borrowing. Di conseguenza questo tipo diventa condivisibile tra threads diversi contemporaneamente senza la possibilità che si verifichino race conditions, quindi possiamo imporre semplicemente il vincolo che G^2 implementi Sync se il tipo dell'etichetta dei nodi e il tipo del peso degli archi lo implementano a loro volta.

G^2 fa internamente uso di puntatori, ciò significa che non è thread-safe automaticamente (per quanto detto nel capitolo 1), quindi sapendo che G^2 è thread-safe per via dell'analisi sopra descritta, è necessario forzare l'implementazione tramite codice unsafe:

```
1  // Non contiene niente quindi può implementare sempre i traits.
2  unsafe impl<'id> Send for GgToken<'id> {}
3  unsafe impl<'id> Sync for GgToken<'id> {}
4
5  // Il contenitore e il riferimento al nodo implementano
6  // Send e Sync se i tipi contenuti implementano i traits.
7  unsafe impl<'id, T: Send, G: Send> Send
8      for GenerationalGraph<'id, T, G>  {}
9  unsafe impl<'id, T: Sync, G: Sync>
10     Send for GenerationalGraph<'id, T, G>  {}
11
12 unsafe impl<'a, 'id, 'b, T: Send, G: Send> Send
13     for NodeRef<'a, 'id, 'b, T, G> {}
14 unsafe impl<'a, 'id, 'b, T: Sync, G: Sync> Sync
15     for NodeRef<'a, 'id, 'b, T, G> {}
```

Nel caso del token, poiché non contiene nessun campo fisico è possibile implementare sempre i trait Send e Sync. Il contenitore e il riferimento al nodo implementano i trait se e solo se il tipo dell'etichetta dei nodi e il tipo del peso degli archi implementano i rispettivi traits, in G^2 non ha come obiettivo quello di rendere thread-safe tipi che non lo sono. Ad esempio Rc non implementa né Send, né Sync e se lo inserissimo come valore implementando Send e Sync, potremmo 'clonare' il reference counter da threads diversi contemporaneamente causando race conditions nell'aggiornamento del contatore interno.

La versione proposta di G^2 consente di implementare gli algoritmi su grafi usando sia codice safe (sempre), sia usando codice unsafe in modo da poter garantire la memory safety seguendo alcune regole. Questo è vero se G^2 può essere utilizzato solo in ambito single-thread, quindi se non implementiamo i trait Send e Sync, mentre se decidiamo di implementarli per l'utilizzo multi-thread, gli algoritmi possono essere implementati solo tramite codice safe (in modo da garantire la memory safety). Il problema è causato dall'assenza di meccanismi di sincronizzazione interni e l'utilizzo di codice unsafe per accedere ai nodi per implementare gli algoritmi su grafi non vieta a altri threads di accedere ai propri nodi durante l'esecuzione di un algoritmo con permessi non coerenti con la regola AXM. Per poter estendere G^2 con i vari algoritmi anche con codice unsafe garantendo l'assenza di race conditions e il rispetto della regola AXM in ambito multi-thread (nei casi in cui sono necessarie buone performance), è possibile creare una seconda versione dello smart pointer che richiede l'utilizzo di un unico token per accedere all'intera struttura, ma a quel punto torneremmo ad avere quasi gli stessi svantaggi offerti da GhostCell. Un'altra alternativa è quella di definire dei metodi unsafe che implementano gli algoritmi tramite codice unsafe in modo da delegare all'utilizzatore la gestione della possibile creazione di race conditions (ad esempio utilizzando una lock esterna).

5.4 Esempi di possibile utilizzo

Creazione (Single-thread)

Di seguito riportiamo un piccolo esempio di possibile utilizzo dello smart pointer in ambito single-thread. Per mostrare anche che non si possono creare dangling references o in generale mostrare che la memory safety è garantita, includiamo alcuni esempi di comandi (commentati con tre punti interrogativi '???'), che se inseriti causerebbero un errore a tempo di compilazione insieme a un piccolo commento sull'errore:

```
1 // Versione dello smart pointer che non implementa Send e Sync.
2 // Non potendolo utilizzare in ambito multi-thread, è possibile
3 // implementare gli algoritmi su grafi tramite estensione unsafe
4 // rispettando il vincolo che la chiusura per eseguire del codice
```

```
5  // su ogni nodo abbia accesso in sola lettura all'ambiente catturato.
6  GenerationalGraph::new(|graph1, mut token1| {
7      let mut x1 = graph1.add(1, &mut token1);
8      let mut x2 = graph1.add(2, &mut token1);
9
10     x1.link(&x2, 1);
11     // Altre generazioni si creano invocando di nuovo il metodo new
12     GenerationalGraph::new(|graph2, mut token2| {
13         let mut y1 = graph2.add(1, &mut token2);
14         let mut y2 = graph2.add(2, &mut token2);
15         y1.link_outer(&x2, 2);
16
17         // Bfs implementata tramite unsafe.
18         y1.bfs(|| {}, |node_val, supp_struct| {
19             println!("Node val: {}", node_val);
20         });
21
22         x1.link_inner(&y1, 1, |from| {
23             // Riusiamo il nodo x1 per effettuare altri link
24             from.link_inner(&y2, 1, || {
25                 println!("{}", *from);
26             });
27             // Vietato rilasciare la memoria. Un rifermento
28             // collegato a graph2 (Y1) è in utilizzo
29             // drop(graph2) ???
30         });
31
32         // Il grafo graph1 è già catturato tramite borrow su (x2).
33         // Non si può catturare anche tramite ownership.
34         //drop(graph1); ???
35     });
36
```

```
37      // È possibile continuare a utilizzare i nodi anche dopo
38      // che il grafo più interno che era in comunicazione con
39      // quello esterno tramite link_inner è stato distrutto.
40      // Cosa non possibile con l'implementazione con GhostCell.
41      x2.link(&x1, 1);
42      println!("{}", *x2);
43  });
```

Notiamo che l'utilizzo delle chiusure in questo modo permette sia di regolare il rilascio della memoria dei grafi (vietandone il rilascio mentre sono attivi dei collegamenti con altri grafi), sia di avere un lifetime diverso per ogni grafo e quindi permette di definire metodi diversi per ogni caso e di controllare a tempo di compilazione che il metodo utilizzato sia coerente con il nodo utilizzato per creare o rimuovere un arco all'interno della stessa generazione o tra generazioni diverse.

Creazione (Multi-thread)

Di seguito riportiamo un piccolo esempio di possibile utilizzo di G^2 in ambito multi-thread usando la libreria rayon per effettuare computazioni con il parallelismo di tipo fork-join:

```
1  // Versione dello smart pointer ce implementa Send e Sync.
2  // Potendolo usare in ambito multi-thread non è possibile
3  // definire gli algoritmi tramite codice unsafe in modo da
4  // garantire la memory safety.
5  GenerationalGraph::new(|graph1, mut token1| {
6      let mut x1 = graph1.add(1, &mut token1);
7      let mut x2 = graph1.add(2, &mut token1);
8
9      x1.link(&x2, 1);
10
11     GenerationalGraph::new(|graph2, mut token2| {
12         let mut y1 = graph2.add(1, &mut token2);
13         let mut y2 = graph2.add(2, &mut token2);
14
15         y1.link(&y2, 2);
```

```
16          y1.link_outer(&x2, 2);
17
18          rayon::join(|| {
19              /* implementazione della bfs safe */
20              println!("{}", *x1);
21          }, || {
22              /* implementazione della bfs safe */
23              println!("{}", *y1);
24
25              // Non si può catturare tramite ownership. Il
26              // borrow (y1) è usato successivamente.
27              // drop(graph2); ???
28          });
29
30          println!("{}", *y1); // Utilizzo di (y1).
31
32          // Si può usare l'algoritmo BFS, ma il metodo deve
33          // essere marcato come unsafe e deve essere invocato
34          // all'interno di un blocco unsafe. Usando unsafe sta
35          // al programmatore verificare che l'invocazione sia corretta
36          unsafe {
37              // Metodo marcato come unsafe!
38              y1.bfs(|| {}, |node_val, supp_struct| {
39                  println!("Node val: {}", node_val);
40              });
41          }
42      });
43      // Si può continuare a utilizzare il grafo
44      // come mostrato nell'esempio precedente.
45      x2.link(&x1, 1);
46      println!("{}", *x2);
47  });
```

La necessità di rayon non deriva da una preferenza, ma da un vincolo imposto dal compilatore, perché i grafi catturati nelle chiusure passate ai thread sono allocati sullo stack di un thread, alcuni grafi dipendono da altri e la loro memoria viene rilasciata al termine del corpo della loro chiusura (vale per entrambe le versioni). Consultando la documentazione per controllare i requisiti della chiusura passata al metodo *spawn* per creare un thread:

```
1  pub fn spawn<F, T>(f: F) -> JoinHandle<T>
2  where
3      F: FnOnce() -> T + Send + 'static,
4      T: Send + 'static { /* ... */ }
```

notiamo che il lifetime della chiusura deve essere **'static**, cioè la chiusura deve essere valida per l'intera durata del programma e questo vale se e solo se tutto l'ambiente catturato ha una validità pari all'intero programma. Questa cosa come detto prima non è vera nel nostro caso, perché i nostri grafi hanno un lifetime inferiore essendo allocati sullo stack, quindi non possiamo condividere l'accesso direttamente. La libreria rayon, invece possiede un metodo join che ha una firma di questo tipo:

```
1  pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
2  where
3      A: FnOnce() -> RA + Send,
4      B: FnOnce() -> RB + Send,
5      RA: Send,
6      RB: Send { /* ... */ }
```

Le chiusure richieste non richiedono un lifetime, perché in questo caso il metodo join sospende il thread chiamante fino a che i threads creati non hanno terminato l'esecuzione e ciò implica che possiamo condividere l'accesso ai grafi, perché rimarranno sicuramente validi per tutta la durata di esecuzione dei threads. Precisiamo che non è possibile utilizzare Arc sui grafi per utilizzare la funzione *thread::spawn*, perché i grafi hanno una validità di chiusura e non possono *scappare* al di fuori di esse (più esternamente), quindi non c'è modo di aumentare il lifetime del grafo a **'static**.

Creazione ricorsiva

Con la nostra implementazione è possibile generare un numero di grafi arbitrario ricorsivamente

specificando i lifetime dei nodi parametro per parametro, in modo che l'analizzatore statico possa accettare nodi che non hanno lo stesso lifetime come parametri della chiamata ricorsiva. Ad esempio in questo modo:

```
fn rec_gen<'a, 'b, 'c, 'd, 'id1, 'id2>(
    out1: &mut NodeRef<'a, 'id1, 'b, i32, i32>,
    out2: &mut NodeRef<'c, 'id2, 'd, i32, i32>) {

    GenerationalGraph::new(|graph, mut token| {
        let mut n = graph.add(1, &mut token);
        **out1 = 0;
        rec_gen(out2, &mut n);
    })
}
```

Non è possibile passare un numero di nodi arbitrario alle generazioni successive, perché (come visto nel capitolo 1) le strutture che permettono di gestire collezioni di valori come Vec e LinkedList richiedono che tutti i nodi abbiano lo stesso lifetime (o siano riducibili a esso) altrimenti si perderebbe ogni informazione sulla relativa durata. Questo banalmente non è possibile, perché i nodi di altri grafi hanno un brand **'id** diverso, infatti se ad esempio proviamo a implementare il passaggio nel modo seguente:

```
fn rec_gen_error<'a, 'id, 'b>(
    outs: &mut LinkedList<NodeRef<'a, 'id, 'b, i32, i32>>) {
    GenerationalGraph::new(|graph, mut token| {
        let mut n = graph.add(1, &mut token);
        outs.push_back(n);
        rec_gen_error(outs);
        outs.pop_front();
    })
}
```

otteniamo un errore di questo tipo:

```
error[E0521]: borrowed data escapes outside of closure
   --> src\main.rs:200:21
    |
198 |     outs: &mut LinkedList<NodeRef<'a, 'id, 'b, i32, i32>>) {
    |     ---- `outs` declared here, outside of the closure body
199 |     GenerationalGraph::new(|graph, mut token| {
    |                             ----- `graph` is a reference that is only
     valid in the closure body
200 |         let mut n = graph.add(1, &mut token);
    |                     ^^^^^^^^^^^^^^^^^^^^^^^^ `graph` escapes the
     closure body here
```

Il questo caso l'errore è dato dal brand che essendo invariante non accetta nodi appartenenti a altri grafi all'interno della lista. Precisiamo che nemmeno facendo trasformazioni per eliminare il brand dei nodi potremmo riuscire a fare questa cosa, il motivo deriva dagli altri lifetime che impongono una regola per i nodi aggiunti successivamente, che impedisce di aggiungere nodi con lifetime minore per via della presenza del lifetime covariante (necessario per sapere la durata di un nodo) e non è possibile nemmeno implementare una struttura che garantisce l'assenza di creazione di dangling references che ignora questi lifetime altrimenti perderemmo ogni informazione sulla loro validità.

5.5 Analisi dello smart pointer

Safety della API

Abbiamo già discusso parte della safety offerta dalla API dello smart pointer durante l'implementazione giustificando alcune scelte, che a livello di codice possono non avere senso per chi utilizza linguaggi più ad alto livello che utilizzano dei garbage collector e altri meccanismi per garantire l'assenza di race conditions e la memory safety.
In generale:

- G^2 sfrutta il normale meccanismo di ownership e borrowing per regolare gli accessi ai nodi sia nel caso single-thread, sia nel caso multi-thread senza costi aggiuntivi.
- G^2 sfrutta il token per regolare gli accessi alla arena per l'allocazione di un nuovo nodo per dimostrare possesso e accesso esclusivo sia nel caso single-thread, sia nel caso multi-thread.
- G^2 Sfrutta le arene e lifetimes per garantire la validità degli archi senza controlli a tempo di esecuzione.

- G^2 Sfrutta i lifetimes per validare i riferimenti ai nodi dei grafi ritornati dai metodi e per controllare la correttezza nell'utilizzo dei metodi di creazione degli archi.

Vantaggi e Svantaggi

L'utilizzo di G^2 per l'implementazione di questa tipologia di grafo porta sia a dei vantaggi, sia a degli svantaggi.

Vantaggi:

- Può essere utilizzato sia in ambito single-thread, sia in ambito multi-thread.
- Offre la possibilità di creare grafi suddivisi in generazioni senza introdurre costi a tempo di esecuzione utilizzando esclusivamente codice safe, in modo da migliorare i costi in termini di tempo e di memoria rispetto a implementare la stessa struttura tramite l'utilizzo di reference counters e locks.
- Offre la possibilità di essere esteso con i vari algoritmi per grafi usando codice safe e codice unsafe in modo da garantire la memory safety se non implementiamo i traits Send e Sync. Nel caso di implementazione dei traits Send e Sync invece è possibile estendere lo smart pointer tramite codice safe garantendo la memory safety, ma non è possibile estenderlo tramite codice unsafe in modo da garantire la memory safety senza modificare la definizione dello smart pointer introducendo dei costi a tempo di esecuzione.

Svantaggi:

- La generazione ricorsiva delle generazioni è comunque limitata dall'analizzatore statico e non permette di passare un numero generico di nodi alle generazioni successive avendo bisogno di esplicitare tutti i lifetime per ogni nodo in entrambe le versioni.
- La deallocazione dei nodi di un grafo non può essere fatta individualmente, perché alcuni archi verso altri grafi non sarebbero più validi e deve essere fatta in blocco quando un grafo non è più utilizzato (cioè al termine del corpo della rispettiva chiusura o anticipatamente tramite drop quando il grafo non è più utilizzato).

- L'utilizzo di lifetime in un contesto diverso da quello normalmente utilizzato del compilatore impone dei vincoli forti di espressività quando si usano altre strutture insieme a questo smart pointer. Ad esempio, i nodi di grafi diversi non possono essere raggruppati in un'unica lista.
- Non possiamo usare la funzione thread::spawn per creare un thread e dobbiamo utilizzare librerie di terze parti come rayon [3, 19] che permettono di eseguire computazioni in parallelo usando il parallelismo di tipo fork-join.

Benchmarks

In questo paragrafo presentiamo dei benchmarks per mettere a confronto le varie implementazioni dello stesso tipo di grafo (diretto, etichettato, pesato, semplice e ciclico) tramite Rc, Arc, GhostCell e GenerationalGraph. Le implementazioni usate per rappresentare i grafi con altri smart pointers sono le seguenti:

```
1  // Per rappresentare gli archi, poiché i reference counter e GhostCell
2  // non possono essere usati come chiavi non avendo una trasformazione
3  // hash, utilizziamo un identificatore intero unico per ogni nodo e
4  // associamo questo identificatore al nodo destinazione
5  // con relativo peso dell'arco (tupla Nodo - Peso).
6
7  struct GraphNodeRc<T, G> {
8      id: i32, // Identificatore
9      val: T, // Etichetta
10     // Mappa: Identificatore nodo -> (Nodo, Peso)
11     links: HashMap<i32, (Rc<RefCell<GraphNodeRc<T, G>>>, G)>
12 }
13
14 struct GraphNodeArc<T, G> {
15     id: i32, // Identificatore
16     val: T, // Etichetta
17     // Mappa: Identificatore nodo -> (Nodo, Peso)
18     links: HashMap<i32, (Arc<Mutex<GraphNodeArc<T, G>>>, G)>
19 }
```

```
20
21  struct GraphNodeGhost<'arena, 'id, T, G> {
22      id: i32, // Identificatore
23      val: T, // Etichetta
24      // Mappa: Identificatore nodo -> (Nodo, Peso)
25      links: HashMap<i32, (&'arena GhostCell<'id,
26          GraphNodeGhost<'arena, 'id, T, G>>, G)>,
27  }
```

Questi benchmarks sono stati eseguiti con la libreria criterion e l'approccio utilizzato è analogo ai benchmarks eseguiti su GhostCell, quindi è stata presa la media aritmetica di 100 prove (iterazioni) secondo i criteri seguenti:

- Il benchmark per l'aggiunta e creazione di archi tra nodi, è stato eseguito creando 100000 nodi e collegandoli in catena come una lista concatenata singola. Questo test è stato eseguito avendo cura di non far terminare lo scope di definizione dei grafi che provocherebbe anche il rilascio della memoria (rimozione di tutti i nodi).
- Il benchmark per la rimozione degli archi è stato eseguito prendendo il grafo ('lista concatenata singola') creato nel punto precedente e sono stati rimossi tutti gli archi sui 100000 nodi creati precedentemente.
- Il benchmark per l'iterazione è stato eseguito effettuando una ricerca in ampiezza sul grafo creato nel primo punto, cioè un grafo a forma di lista concatenata singola con 100000 nodi. Nel caso multi-thread è stata utilizzata la libreria rayon per effettuare iterazioni in parallelo con il parallelismo di tipo fork-join.

	Allocazione di 100000 nodi e Creazione 100000 archi - Grafo vuoto	**Rimozione 100000 archi - Grafo già pronto a forma di lista singola**	**BFS su 100000 nodi - Grafo a forma di lista singola - 1 Thread**	**BFS su 100000 nodi - Grafo a forma di lista singola - 4 Threads**
Grafo - Rc, RefCell	26.30 ms	7.69 ms	17.94 ms	Non compatibile
Grafo - Arc, Mutex	41.62 ms	25.73 ms	39.96 ms	76.65 ms
Grafo - GhostCell	22.42 ms	9.53 ms	13.21 ms	27.30 ms
Grafo - GenerationalGraph	19.68 ms	10.96 ms	15.43 ms - Unsafe / 31.72ms - Safe	29.91 ms - Unsafe / 51.74 ms - Safe

Notiamo che in questo caso a differenza dei benchmarks eseguiti nel capitolo 4, i valori tra Rc

GhostCell e G^2 sono molto simili e l'unico caso in cui la differenza è considerevole è il caso con Arc. Questo accade perché per rappresentare i grafi abbiamo utilizzato lo smart pointer HashMap che introduce costi di inserimento e di ricerca, dato che il numero di successori non è fisso per i vari nodi successori come invece succede per le liste concatenate. Notiamo che G^2 si comporta in modo molto simile a GhostCell nei primi due benchmarks, mentre per i benchmarks sull'iterazione fornisce valori simili a GhostCell nel caso di utilizzo di algoritmi implementati mediante codice unsafe, mentre un risultato intermedio tra GhostCell e Arc nel caso in cui si implementi l'algoritmo mediante codice safe come visto nella sezione 5.3. Precisiamo che le scritture safe e unsafe all'interno della tabella significano rispettivamente versione implementata tramite codice safe e versione implementata tramite codice unsafe.

Conclusioni

La tesi ha affrontato le problematiche relative all'implementazione di smart pointers in Rust e dopo aver analizzato lo smart pointer GhostCell propone lo smart pointer GenerationalGraph (G^2). G^2 è uno smart pointer che consente di creare e manipolare grafi in Rust. G^2 in particolare usa a proprio vantaggio il type system offerto da Rust. Inoltre, G2 è caratterizzato dal fatto di avere bassi costi di esecuzione nell'implementazione di grafi suddivisi in generazioni, garantendo la memory safety senza l'utilizzo di un garbage collector. G^2 offre una API unica sia per l'utilizzo in ambito single-thread, sia per l'utilizzo in ambito multi-thread. Inoltre, è predisposto all'implementazione dei vari algoritmi su grafi tramite codice safe e eventualmente tramite codice unsafe (cioè codice che utilizza alcune operazioni di cui non è possibile garantire la memory safety) sotto specifiche ipotesi.

G^2 attualmente può gestire nodi di un solo tipo. Tuttavia può essere esteso a gestire nodi di tipo diverso tramite una variante del memory allocator utilizzato, permettendo di conseguenza di mettere in comunicazione anche grafi che operano su tipi diversi. Questa estensione tuttavia richiede un vincolo aggiuntivo, ovvero l'utilizzo di descrittori a tempo di esecuzione che permettono di avere delle informazioni sul tipo allocato. L'introduzione dei descrittori comporta alcuni costi aggiuntivi per il controllo sul tipo a tempo di esecuzione.

Nel capitolo 5 si affronta l'implementazione dello smart pointer e ci si limita a giustificare le varie scelte implementative attraverso esempi e intuizioni per convincersi della memory safety in ambito single-thread e multi-thread, ma non è stata fornita una prova formale di correttezza.

Esistono diversi strumenti per verificare le proprietà di un programma Rust. RustBelt è un proof assistant per la verifica di proprietà di codice Rust. RustBelt è stato implementato ed è disponibile come libreria del proof assistant Coq [32][24] ed è stato impiegato con grande efficacia nella dimostrazione di proprietà di safety per Rust per alcuni smart pointers come ad esempio GhostCell. Per verificare la correttezza di un programma Rust esistono anche altri strumenti come Verus [26][27]. Verus è uno strumento di model checking che permette di verificare la memory safety di programmi Rust. Un naturale sviluppo del lavoro presentato in questa tesi consiste nell'uso di uno di questi strumenti di verifica per dimostrare che G^2 garantisce la memory safety. RustBelt è lo strumento

naturale per effettuare la prova di correttezza.

Nell'illustrazione di GhostCell e G^2 è stato presentato anche il concetto di Higher Ranked Trait Bounds (HRTBs) [7] (cioè il costrutto for<>) per la definizione dei lifetimes, anche se non lo abbiamo mai nominato esplicitamente. La nozione di HRTBs è strettamente collegata con le problematiche di polimorfismo parametrico e higher ranked types presenti in linguaggi quali Haskell e analizzata in termini fondazionali dal System F [18]. Non abbiamo approfondito questo aspetto a causa della limitata documentazione presente per Rust. Uno sviluppo interessante dei risultati della tesi è costituito dall'analisi, all'interno di System F della nozione di HRTBs in Rust, in modo da catturare pienamente le nozioni di chiusura e lifetimes.

Bibliografia

[1] Cargo bench. https://doc.rust-lang.org/stable/cargo/commands/. Ultimo accesso: 01/07/2023.

[2] Crate alloc. https://doc.rust-lang.org/alloc/. Ultimo accesso: 25/06/2023.

[3] Crate rayon. https://docs.rs/rayon/latest/rayon/. Ultimo accesso: 23/06/2023.

[4] Crate typed_arena. https://docs.rs/typed-arena/latest/typed_arena/. Ultimo accesso: 23/06/2023.

[5] Criterion. https://github.com/bheisler/criterion.rs. Ultimo accesso: 27/06/2023.

[6] Difference between reference and borrow. https://www.programiz.com/rust/references-and-borrowing. Ultimo accesso: 20/06/2023.

[7] Higher Ranked Trait Bounds. https://rust-lang.github.io/rfcs/0387-higher-ranked-trait-bounds.html. Ultimo accesso: 03/07/2023.

[8] Module std::marker. https://doc.rust-lang.org/std/marker/index.html. Ultimo accesso: 23/06/2023.

[9] Module std::mem. https://doc.rust-lang.org/std/mem/fn.drop.html. Ultimo accesso: 27/06/2023.

[10] Module std::sync. https://doc.rust-lang.org/std/sync/index.html. Ultimo accesso: 23/06/2023.

[11] Module std::thread. https://doc.rust-lang.org/std/thread/. Ultimo accesso: 23/06/2023.

[12] Parametric polymorphism. https://en.wikipedia.org/wiki/Parametric_polymorphism. Ultimo accesso: 05/07/2023.

[13] Resource acquisition is initialization (RAII). https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization. Ultimo accesso: 20/06/2023.

[14] Rust by Example. https://doc.rust-lang.org/rust-by-example/. Ultimo accesso: 16/06/2023.

[15] The Rust Reference. https://doc.rust-lang.org/reference/. Ultimo accesso: 23/06/2023.

[16] The Rustonomicon. https://doc.rust-lang.org/nomicon/. Ultimo accesso: 16/06/2023.

[17] Smart pointer. https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization. Ultimo accesso: 16/06/2023.

[18] System F. https://en.wikipedia.org/wiki/System_F. Ultimo accesso: 05/07/2023.

[19] Färnstrand and Linus. Parallelization in Rust with fork-join and friends: Creating the fork-join framework. 2019.

[20] Francis Gagné. Difference between a pointer and a reference in Rust. https://stackoverflow.com/questions/62232753. Ultimo accesso: 20/06/2023.

[21] Jon Gjengset. Crust of Rust: Subtyping and variance. https://www.youtube.com/watch?v=iVYWDIW71jk. Ultimo accesso: 23/06/2023.

[22] Manish Goregaokar. Arenas in Rust. https://manishearth.github.io/blog/2021/03/15/arenas-in-rust/. Ultimo accesso: 20/06/2023.

[23] Ralf Jung. Ghostcell. https://gitlab.mpi-sws.org/FP/ghostcell. Ultimo accesso: 23/06/2023.

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in rust. *Commun. ACM*, 64(4):144–152, 2021.

[25] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2023.

[26] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023.

[27] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types (extended version). *CoRR*, abs/2303.05491, 2023.

[28] Matthieu M. What are move semantics in Rust? `https://stackoverflow.com/questions/30288782`. Ultimo accesso: 20/06/2023.

[29] Tim McNamara. Implementing Doubly-Linked Lists in Rust. `https://www.youtube.com/watch?v=C5BiKWvGYw8`. Ultimo accesso: 23/06/2023.

[30] Mae Milano, Joshua Turcotti, and Andrew C. Myers. A flexible type system for fearless concurrency. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 458–473. ACM, 2022.

[31] oli_obk. How does Rust provide move semantics. `https://stackoverflow.com/questions/29490670`. Ultimo accesso: 20/06/2023.

[32] Kenneth Roe and Scott F. Smith. Using the coq theorem prover to verify complex data structure invariants. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 118–121. ACM, 2017.

[33] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: separating permissions from data in Rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.