Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

rust-cc: un cycle collector per programmi scritti nel linguaggio Rust

Relatore:
Chiar.mo Prof.
Saverio Giallorenzo

Presentata da:
Francesco Goretti

I Sessione
Anno Accademico 2023/2024

A mia nonna Caterina...

Sommario

Rust è un recente linguaggio di programmazione che permette, grazie al suo modello di ownership e al ricco type system, di garantire la sicurezza nella gestione della memoria a tempo di compilazione. Un esempio di come tali feature possano essere utilizzate per creare astrazioni utili sono gli smart pointer `Rc` e `Arc` della libreria standard, che offrono puntatori reference counted per realizzare ownership condivisa. Tuttavia, essi soffrono i problemi classici dell'approccio reference counted, principalmente l'incapacità di liberare i cicli di referenze. In questa tesi viene quindi presentato `rust-cc`, un crate scritto nel linguaggio Rust che offre lo smart pointer `Cc` (Cycle Collected), ovvero un puntatore reference counted con la capacità di liberare i cicli di referenze grazie ad un algoritmo di garbage collection basato su cycle collection.

Parole chiave: garbage collection, cycle collection, reference counting, Rust.

Indice

1 Introduzione **1**

2 Nozioni preliminari **3**

2.1 Garbage collection 3

2.1.1 Reference counting 4

2.1.2 Cicli di referenze e cycle collection 4

2.1.3 Mark and sweep 5

2.1.4 Arena 6

2.2 Finalizzazione 7

2.3 Weak pointer 7

2.4 Cleaner 7

3 API **9**

3.1 Feature disponibili 9

3.2 Lo smart pointer `Cc` 10

3.3 Trait `Trace` 11

3.3.1 Condizioni di safety del trait `Trace` 12

3.3.2 Macro derivativa `Trace` 13

3.4 Finalizzazione e trait `Finalize` 15

3.4.1 Macro derivativa `Finalize` 16

3.5 Esempio di creazione di un ciclo di referenze 16

3.6 Weak pointer 17

3.7 Cleaner 19

3.8 State 20

3.9 Collezionamento automatico dei cicli 21
3.9.1 Config 21
3.10 Supporto a target no-std 21

4 Algoritmo di collezionamento **23**
4.1 Layout degli oggetti allocati 23
4.2 Collezionamento in assenza di cicli 25
4.3 Collezionamento in presenza di cicli 25
4.3.1 Individuamento dei cicli spazzatura 26
4.3.2 Analisi dell'identificazione dei cicli 28
4.3.3 Proprietà di correttezza 31
4.3.4 Proprietà di completezza 31
4.4 Finalizzazione 32

5 Implementazione **35**
5.1 Smart pointer `Cc` e il tipo `CcBox` 35
5.1.1 Implementazione delle liste intrusive 36
5.1.2 Puntatore alla vtable 37
5.1.3 CounterMarker 37
5.1.4 Gestione dell'aliasing dei puntatori 37
5.2 Implementazione del collezionamento 38
5.2.1 Identificazione dei cicli 38
5.2.2 Attivazione del collezionamento automatico 38
5.2.3 Note sulle proprietà di correttezza e completezza 40
5.3 Implementazione della finalizzazione 40
5.4 Liberamento della memoria 43
5.5 Weak pointer 44
5.5.1 `Weak e Weakable` 44
5.5.2 Upgrade e downgrade di weak pointer 46
5.6 Cleaner 47

6 Valutazione **51**
6.1 Testing 51
6.2 Comparazione con le alternative 51

6.2.1 gc . . . 52
6.2.2 bacon-rajan-cc . . . 53
6.2.3 safe-gc . . . 54
6.2.4 broom . . . 56
6.3 Benchmarks . . . 57
6.3.1 Stress test . . . 57
6.3.2 Binary trees . . . 58
6.3.3 Binary trees with parent pointers . . . 59
6.3.4 Large linked list . . . 61

7 Conclusioni **63**
7.1 Lavori futuri . . . 64

Appendici

Appendice A Approfondimento sull'incompatibilità tra crate causata da supertrait richiesti solamente con specifiche feature abilitate

Appendice B Casi limite della finalizzazione

Elenco dei codici

2.1 Esempio di ciclo di referenze. 4
3.1 Esempio di utilizzo dello smart pointer `Cc`. 11
3.2 Definizione del trait `Trace`. 11
3.3 Esempio di utilizzo della macro derivativa `Trace`. 13
3.4 Esempio di utilizzo dell'attributo `#[rust_cc(ignore)]`. 14
3.5 Esempio di utilizzo dell'attributo `#[rust_cc(unsafe_no_drop)]`. . . 14
3.6 Definizione del trait `Finalize`. 15
3.7 Esempio di utilizzo della macro derivativa `Finalize`. 16
3.8 Esempio di tipo che permette un ciclo di referenze. 16
3.9 Esempio di creazione di un ciclo di referenze. 17
3.10 Definizione dell'alias `WeakableCc`. 18
3.11 Esempio di utilizzo dei weak pointer. 18
3.12 Esempio di utilizzo di `new_cyclic`. 19
3.13 Esempio di utilizzo dei cleaner. 20
3.14 Segnatura della funzione `config`. 21
5.1 Definizione dello smart pointer `Cc`. 35
5.2 Definizione della struct `CcBox`. 36
5.3 Esempio di finalizzazione con upgrade di un weak pointer. 41
5.4 Esempio di finalizzazione ricorsiva in assenza di cicli. 43
5.5 Definizione dello smart pointer `Weak`. 45
5.6 Definizione della struct `Weakable`. 46
5.7 Definizione della funzione `Weak::upgrade`. 46
5.8 Definizione di `CleanerFn`. 48
6.1 Esempio di utilizzo del crate `gc`. 53

6.2 Esempio di utilizzo del crate `bacon-rajan-cc`. 54
6.3 Esempio di utilizzo del crate `safe-gc`. 55
6.4 Esempio di utilizzo del crate `broom`. 57
B.1 Esempio di caso limite della finalizzazione. . . . . .

Elenco delle figure

4.1 Layout di un oggetto allocato di tipo `T`. 24
4.2 Layout di `CounterMarker` 24
4.3 Esempio di collezionamento di un ciclo di referenze. 26
4.4 Diagramma degli stati di un oggetto. 28
4.5 Esempio di esecuzione dell'algoritmo di collezionamento. 30
5.1 Layout di `WeakMetadata` 45
6.1 Diagramma a violino dei risultati di `stress test`. 58
6.2 Diagramma a violino dei risultati di `binary trees`. 59
6.3 Diagramma a violino dei risultati di `binary trees with parent pointers`. 60
6.4 Diagramma a violino dei risultati di `large linked list`. 61
A.1 Albero delle dipendenze del crate `my-crate`. . . . . .

Elenco delle tabelle

6.1 Risultati di `stress test`. 58
6.2 Risultati di `binary trees`. 59
6.3 Risultati di `binary trees with parent pointers`. 60
6.4 Risultati di `large linked list`. 61

Capitolo 1

Introduzione

Rust [17] è un linguaggio di programmazione recente che ha raggiunto la prima versione stabile nel maggio 2015. È principalmente conosciuto per il suo ricco type system e il modello di ownership grazie ai quali riesce, senza necessitare di un garbage collector, a garantire la sicurezza nella gestione della memoria e per cui individua molte tipologie di bug già a tempo di compilazione.

Rust riesce nel suo intento proprio grazie a queste feature, che sono infatti utilizzate dalle librerie per creare API safe, ovvero che non possono mai causare comportamento non definito, anche in caso di un loro uso erroneo. Ne è un primo esempio la libreria standard, che fornisce molte strutture dati fondamentali per ogni programma Rust.

Per rendere possibile scrivere programmi dove un valore può avere più proprietari, Rust introduce il concetto di ownership condivisa. Lo smart pointer `Rc` e la sua versione thread-safe `Arc` della libreria standard sono due esempi di tipi che permettono di condividere il possesso di un valore. La loro implementazione è basata sul reference counting e quindi soffre della incapacità di liberare i cicli di referenze. Questo li rende poco utili in caso di strutture dati intrinsecamente cicliche come i grafi oppure in ambienti nei quali non si conosca precisamente l'ownership dei propri dati a priori, ad esempio all'interno di macchine virtuali. In questi casi, è necessario impiegare una qualche forma di gestione automatica della memoria che non soffra dei limiti

imposti dal reference counting, ad esempio utilizzando un garbage collector.

A questo scopo, in questa tesi verrà presentato `rust-cc`, un crate open-source scritto nel linguaggio Rust, pubblicato su crates.io all'indirizzo `https://crates.io/crates/rust-cc` e i cui sorgenti sono disponibili nel repository GitHub dedicato `https://github.com/frengor/rust-cc`, che fornisce il puntatore reference counted `Cc` (Cycle Collected), capace di collezionare i cicli di referenze tramite l'implementazione di un algoritmo efficiente di garbage collection basato su cycle collection, ovvero che mira a collezionare i cicli di referenze spazzatura, in particolare sfruttando la conoscenza dei reference counter.

Sarà inoltre mostrato come `rust-cc` presenti una API ergonomica e completa che fa un uso minimale di unsafe, delineandone sempre in maniera precisa i contratti da soddisfare, e che sia utilizzabile completamente in maniera safe, anche grazie alla presenza di macro derivative, in particolar modo per il trait `Trace`.

Infine, ne verrà effettuata una valutazione sia qualitativa che quantitativa attraverso la comparazione con alcuni garbage collector alternativi presenti nell'ecosistema Rust.

Capitolo 2

Nozioni preliminari

In questo capitolo verranno introdotti i concetti relativi alla garbage collection utili alla comprensione della tesi. Inoltre, per la natura di questa tesi, si presume che il lettore abbia già una conoscenza preliminare del linguaggio di programmazione Rust e del gestore di pacchetti Cargo.

2.1 Garbage collection

Per *garbage collection* si intende un meccanismo di gestione automatica della memoria e gli algoritmi che implementano tale meccanismo vengono chiamati *garbage collector*. In particolare, essi identificano le allocazioni di memoria non più raggiungibili dal programma e ne gestiscono la liberazione.

Definizione (Allocazione raggiungibile)**.** Una allocazione di memoria si dice *raggiungibile dal programma* (oppure *in uso* o *viva*) se è ancora accessibile dal programma, ovvero se è possibile che venga acceduta tramite il dereferenziamento di un puntatore da parte del programma.
Inoltre, tali puntatori vengono chiamati *puntatori forti* o *strong pointer*.

Nel corso degli anni sono stati inventati numerosi algoritmi di garbage collection, di cui verranno riportate, nel seguito, le tipologie più importanti.

2.1.1 Reference counting

La tecnica più semplice per l'implementazione di un meccanismo di garbage collection è il reference counting[3]: viene mantenuto, per ogni allocazione, il numero di referenze ad essa. Ogni volta che viene creata una nuova referenza il reference counter viene incrementato, mentre viene decrementato quando una referenza viene distrutta. Quando il reference counter raggiunge zero l'allocazione viene liberata in quanto, non esistendo più referenze ad essa, risulterà non raggiungibile.

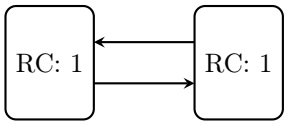
I vantaggi principali della tecnica di reference counting sono di essere semplice da implementare e di non richiedere lunghe pause al programma, oltre al permettere di liberare la memoria nel momento esatto in cui essa diventa inutilizzata.

Tuttavia, il reference counting presenta anche due svantaggi importanti:

1. ogni volta che viene creata o distrutta una referenza deve essere effettuato un incremento o un decremento, che può diventare costoso all'aumentare del numero di referenze;
2. i cicli di referenze non sono collezionati.

2.1.2 Cicli di referenze e cycle collection

Come anticipato nel paragrafo precedente, i cicli di referenze non vengono collezionati da parte degli algoritmi di reference counting[11]. Infatti, come mostrato in figura 2.1, il reference counter di due allocazioni facenti parte di un ciclo di referenze (non raggiungibile) non scende mai a zero ed è quindi impossibile collezionarne i membri.



Codice 2.1: Esempio di ciclo di referenze con relativi reference counter.

Per risolvere questo problema sono stati inventati particolari garbage collector chiamati cycle collector, come quello presentato in questa tesi, che collezionano i cicli di referenze spazzatura.

Definizione (Cicli spazzatura)**.** Un ciclo di referenze si dice *spazzatura* quando tutte le referenze a membri del ciclo sono interne al ciclo stesso.

Proprietà. Se un ciclo di referenze è spazzatura allora tutti i suoi membri sono non raggiungibili.

Dimostrazione. Se un ciclo di referenze è spazzatura, allora le uniche referenze a membri del ciclo possono essere solamente interne al ciclo stesso e quindi essi risultano non raggiungibili. □

2.1.3 Mark and sweep

La tecnica mark and sweep, inventata nel 1960 da John McCarthy per il linguaggio Lisp[12], si basa sull'esecuzione di due fasi di tracciamento, chiamate mark e sweep, al fine di individuare quali allocazioni siano raggiungibili e quali invece sia possibile liberare.

L'idea alla base dell'algoritmo è quindi di utilizzare la definizione di raggiungibilità per individuare le allocazioni ancora vive e di determinare quelle da liberare per differenza con la totalità della memoria allocata.

Definizione (Insieme delle radici)**.** Viene chiamato *insieme delle radici* l'insieme delle allocazioni puntate da puntatori presenti:

1. nella memoria stack;
2. nei registri;
3. nelle costanti;
4. nelle variabili globali.

L'insieme delle radici è quindi l'insieme minimo delle allocazioni sicuramente vive in quanto accessibili al programma.

Durante la fase di mark vengono marcate tutte le allocazioni raggiunte a partire dall'insieme delle radici seguendone ricorsivamente i puntatori contenuti, individuando quindi tutte le allocazioni ancora vive. La fase di sweep, invece, determina la me-

moria da liberare effettuando una scansione dello heap e liberando le allocazioni non precedentemente marcate.

La tecnica mark and sweep è molto potente e offre diversi vantaggi rispetto al reference counting, come il non dover mantenere il reference counter e collezionare i cicli di referenze.

Tuttavia, presenta anche alcuni svantaggi:

1. la memoria non viene più liberata nel momento in cui essa diventa inutilizzata;
2. richiede l'interruzione dell'esecuzione del programma per l'esecuzione della garbage collection.

Infine, implementare un garbage collector mark and sweep per programmi scritti nel linguaggio Rust presenta alcune sfide importanti:

1. Rust non fornisce alcun supporto all'individuazione dell'insieme delle radici;
2. ad ogni passo della fase di mark, *tutti* i puntatori contenuti nelle allocazioni devono essere tracciati, che può però risultare complicato dal momento che Rust non offre meccanismi per effettuare tracciamenti.

2.1.4 Arena

Una arena o regione è una collezione di allocazioni soggette ad una politica di gestione comune. In particolare, quando l'arena viene distrutta, viene sempre liberata tutta la relativa memoria gestita.

Due delle politiche di gestione comunemente utilizzate sono:

1. *bumping*[9], che alloca la memoria richiesta a partire dalla prima locazione libera di un blocco contiguo di memoria ottenuto alla creazione dell'arena o all'esaurimento del blocco precedente. Tale approccio rende molto veloce creare nuove allocazioni, ma sacrifica la possibilità di deallocare singole allocazioni;
2. *garbage collection*, ovvero viene impiegato un garbage collector per liberare le allocazioni non più in uso all'interno dell'arena. Il vantaggio di questo ap-

proccio risiede principalmente nel non dover effettuare una scansione completa dello heap, ma solamente delle allocazioni gestite dall'arena.

2.2 Finalizzazione

Un garbage collector che supporta la finalizzazione permette di eseguire, prima di liberare una allocazione di memoria, una funzione chiamata finalizzatore. Si permettono così azioni di pulizia supplementari, come la chiusura di un file o l'azzeramento di un'area di memoria contenente dati sensibili.

Inoltre, solitamente i finalizzatori hanno accesso alla allocazione che sta venendo finalizzata e hanno quindi la possibilità di resuscitarla:

Definizione (Allocazione resuscitata)**.** Una allocazione si dice *resuscitata* se era stata determinata essere spazzatura, ma successivamente all'esecuzione di un finalizzatore risulta di nuovo raggiungibile da parte del programma.

Perciò, i garbage collector che supportano la finalizzazione presentano anche delle misure per evitare di liberare allocazioni resuscitate.

2.3 Weak pointer

I weak pointer, o puntatori deboli, sono puntatori che non impediscono il collezionamento dell'allocazione puntata. Proprio per questo, non sono direttamente dereferenziabili, ma richiedono di essere prima trasformati in uno strong pointer. Tale conversione può ovviamente fallire, ad esempio nel caso l'allocazione puntata sia già stata liberata.

2.4 Cleaner

I cleaner, introdotti nel linguaggio Java a partire dalla versione 9 [13], sono un'alternativa al meccanismo della finalizzazione che ne fornisce una funzionalità simile, ma rendendo impossibile resuscitare gli oggetti.

Una volta creato un `Cleaner`, è possibile associare ad ogni oggetto una o più espressioni lambda, chiamate pulitori, che verranno eseguite quando l'oggetto a cui sono associate viene liberato. Inoltre, è importante assicurarsi che non venga catturata, dalla espressione lambda, nessuna referenza all'oggetto a cui è associata, altrimenti l'oggetto non diventerà mai spazzatura e non sarà mai liberato, essendo referenziato dal cleaner.

Capitolo 3

API

In questo capitolo viene presentata l'Application Programming Interface (API) di rust-cc, insieme ad alcuni esempi di utilizzo della libreria.

3.1 Feature disponibili

rust-cc è completamente modulare ed ogni feature è abilitabile in base alle esigenze di utilizzo della libreria. Di seguito è riportata la lista delle feature presenti:

- `finalization`: Abilita la finalizzazione;
- `auto-collect`: Abilita l'esecuzione periodica automatica dei collezionamenti dei cicli;
- `weak-ptr`: Abilita i weak pointer;
- `cleaners`: Abilita i cleaner;
- `derive`: Abilita le macro procedurali per derivare automaticamente i trait `Trace` e `Finalize`;
- `std`: Abilitata il supporto alla libreria standard, vedi 3.10.

- `nightly`: Abilita il supporto alle feature in prova e non ancora stabilizzate di Rust presenti sulla toolchain NIGHTLY. È richiesta se la feature `std` non è abilitata.

Di default sono abilitate le feature `auto-collect`, `finalization`, `derive` e `std`.

3.2 Lo smart pointer `Cc`

Lo smart pointer `Cc<T>` è il tipo fondamentale di rust-cc e serve ad allocare ed interagire con la memoria gestita dal cycle collector. Si comporta come un puntatore reference counted (`Rc`) che libera automaticamente i cicli di referenze non più in uso dal programma.

Una volta allocata memoria utilizzando la funzione associata `Cc::new`, essa verrà liberata immediatamente quando il numero di referenze ad essa scenderà a zero oppure, se fa parte di un ciclo di referenze, durante una esecuzione del collezionamento quando il ciclo non sarà più utilizzato dal programma.

Inoltre, gli smart pointer `Cc` non implementano `Copy`, ma possono essere clonati utilizzando il metodo `clone`, che incrementa il contatore delle referenze e ritorna un nuovo puntatore `Cc` alla locazione puntata.

Implementano anche il trait `Deref`, che permette di dereferenziare ogni istanza di un `Cc` in una referenza immutabile al valore puntato, e i trait `Trace` e `Finalize` per permettere il collezionamento (vedi 3.3 e 3.4).

Non implementano, invece, i trait `Send` e `Sync`, in quanto non sono thread-safe.

Infine, alcuni metodi importanti e che verranno citati nel seguito sono:

- `strong_count`: ritorna il numero totale di puntatori `Cc` all'allocazione puntata;
- `finalize_again` e `already_finalized`: due metodi legati alla finalizzazione (vedi 3.4);
- `mark_alive`: un metodo di basso livello per marcare l'allocazione puntata come non da analizzare in quanto ancora utilizzata (vedi 4.3). Si noti che anche clonare un `Cc` marca l'allocazione come non da analizzare.

```rust
let string: Cc<String> = Cc::new(String::from("a-string"));
let string2: Cc<String> = string.clone();

// string e string2 puntano alla stessa stringa.
// Possiamo verificarlo dereferenziando le due variabili
// e comparandole per contenuto e per indirizzo
assert_eq!(&*string, &*string2);
assert!(std::ptr::eq(&*string, &*string2));
```

Codice 3.1: Esempio di utilizzo dello smart pointer `Cc`.

3.3 Trait `Trace`

Per permettere il collezionamento, ogni tipo collezionabile deve implementare il trait `Trace`. Lo scopo di tale trait è di permettere il tracciamento dei puntatori `Cc` salvati nei campi degli tipi collezionabili.

Molti tipi della libreria standard implementano già `Trace` e sono quindi allocabili utilizzando `Cc::new`, come i tipi numerici, il tipo unit `()`, `String`, `Box`, `RefCell` (ma non `Cell`), ecc.

La definizione di `Trace` è riportata nel codice 3.2. Il metodo `Trace::trace`, quando invocato, ha lo scopo di tracciare (ovvero chiamare `trace` su) gli smart pointer `Cc` contenuti nell'istanza sulla quale è stato chiamato (`self`).

```rust
pub unsafe trait Trace: Finalize {
    fn trace(&self, ctx: &mut Context<'_>);
}
```

Codice 3.2: Definizione del trait `Trace`.

La struct `Context` contiene informazioni legate al contesto del tracciamento e non presenta nè metodi nè campi pubblici, in quanto acceduta soltanto internamente dall'implementazione del collezionatore. Passare le informazioni relative al contesto dietro referenza mutabile adempie anche allo scopo, non avendo costruttori pubblici, di rendere impossibile chiamare il metodo `trace` al di fuori di un tracciamento.

`Trace` è dichiarato come unsafe da implementare, in quanto alcune condizioni sul tracciamento non possono essere espresse all'interno del type system di Rust e sono lasciate da verificare al programmatore. Tali condizioni sono presentate nel prossimo paragrafo.

3.3.1 Condizioni di safety del trait `Trace`

Prima di presentare le condizioni di safety del trait `Trace`, è necessario presentare il concetto di fase di tracciamento: ogni invocazione del metodo `trace` avviene durante una fase di tracciamento, che è una delle fasi dell'algoritmo di collezionamento dei cicli di referenze (vedi 4.3). Le fasi di tracciamento possono essere individuate utilizzando la funzione `state::is_tracing` (vedi 3.8), che ritorna `true` solo durante un tracciamento.

Inoltre, due chiamate di `trace` si dicono appartenenti alla stessa fase di tracciamento se `state::is_tracing()` non ha mai ritornato `false` tra le due invocazioni (di `trace`).

Presentiamo ora le condizioni di safety del trait `Trace`:

- l'implementazione del metodo `trace` può tracciare, al massimo una volta, ogni istanza di uno smart pointer `Cc` posseduta esclusivamente dall'istanza sulla quale è stato invocato il metodo (`self`). Nessun altro smart pointer `Cc` può essere tracciato;
- è sempre possibile (ed è sempre safe) far avvenire un panic;
- due chiamate al metodo `trace` sullo stesso valore che appartengono alla stessa fase di tracciamento devono comportarsi "allo stesso modo", ovvero devono tracciare gli stessi `Cc`. Se avviene un panic durante la seconda di tali chiamate al metodo `trace` ma non nella prima, allora l'insieme delle istanze di `Cc` tracciate nella seconda deve essere un sottoinsieme di quelle tracciate nella prima;
- l'implementazione del metodo `trace` non può creare, clonare, dereferenziare o distruggere nessuno smart pointer `Cc`;

- se il tipo implementante `Trace` implementa il trait `Drop`, allora l'implementazione di `Drop::drop` non può creare, clonare, muovere, dereferenziare, distruggere o chiamare nessun metodo su nessuna istanza di uno smart pointer `Cc`.

È quindi safe non tracciare mai alcuni campi. Avere la possibilità di non tracciare alcuni campi espone alla possibilità di memory leak, in quanto un `Cc` non tracciato non permetterà il collezionamento dell'allocazione puntata. Tuttavia, avere tale possibilità è importante, siccome i tipi dichiarati in altre librerie (ed anche alcuni tipi appartenenti alla libreria standard, ad esempio `Cell`) non implementano `Trace`.

3.3.2 Macro derivativa `Trace`

Per permettere l'implementazione di `Trace` anche da parte di codice safe, che non può utilizzare l'unsafe, esiste la macro derivativa `#[derive(Trace)]`. Tale macro si occupa di generare l'implementazione del trait automaticamente e di assicurarsi che tutte le condizioni di safety siano soddisfatte.

Un esempio di utilizzo della macro è presente nel codice 3.3. La macro traccia ogni campo chiamando una volta il metodo `trace` su ognuno di essi.

```
#[derive(Trace)]
struct Foo<T: Trace> {
    foo: u32,
    bar: Bar<T>,
}
```

Codice 3.3: Esempio di utilizzo della macro derivativa `Trace`.

È possibile evitare che un campo venga tracciato tramite l'utilizzo dell'attributo `#[rust_cc(ignore)]`, come si può vedere nell'esempio riportato nel codice 3.4.

Inoltre, per rispettare le condizioni di safety di `Trace` riguardanti i distruttori, viene sempre generata l'implementazione di un distruttore vuoto in aggiunta all'implementazione del trait `Trace`.

```rust
#[derive(Trace)]
struct Foo<T: Trace> {
    bar: Cc<T>,
    #[rust_cc(ignore)]
    cell: Cell<u32>,
}
```

Codice 3.4: Esempio di utilizzo dell'attributo `#[rust_cc(ignore)]`. Il campo `bar` sarà tracciato, mentre `cell` verrà ignorato.

Generare tale implementazione aggiuntiva è necessario, in quanto la macro non può verificare, durante la sua esecuzione, nè la presenza di un distruttore preesistente, nè se tale distruttore rispetti le condizioni di safety.

Viene quindi generata preventivamente una implementazione vuota, in maniera da evitare che possa venire implementato un distruttore unsound da parte di codice safe.

È possibile evitare che la macro generi l'implementazione del distruttore, utilizzando l'attributo `#[rust_cc(unsafe_no_drop)]`, come mostrato nel codice 3.5. L'uso di tale attributo è ovviamente da considerarsi unsafe, in quanto permette l'implementazione di un distruttore personalizzato e deve quindi rispettare le condizioni di safety del trait `Trace`.

```rust
#[derive(Trace)]
#[rust_cc(unsafe_no_drop)]
struct Foo<T: Trace> {
    bar: Cc<T>,
}

// É ora possibile implementare un distruttore personalizzato
impl<T: Trace> Drop for Foo<T> {
    ...
}
```

Codice 3.5: Esempio di utilizzo dell'attributo `#[rust_cc(unsafe_no_drop)]`.

3.4 Finalizzazione e trait `Finalize`

rust-cc supporta la finalizzazione tramite il trait `Finalize`, la cui definizione è riportata nel codice 3.6.

```
pub trait Finalize {
    fn finalize(&self) {
    }
}
```

Codice 3.6: Definizione del trait `Finalize`.

Il metodo `Finalize::finalize` è il finalizzatore, che verrà chiamato automaticamente una sola volta per ogni valore allocato. Per verificare se il finalizzatore sia già stato eseguito si può usare il metodo `Cc::already_finalized`, mentre in caso si voglia far eseguire nuovamente il finalizzatore si può usare il metodo `Cc::finalize_again` (si noti che tale metodo non è chiamabile durante l'esecuzione di un finalizzatore). Inoltre, ogni valore allocato durante l'esecuzione di un finalizzatore è considerato come già finalizzato e quindi il suo finalizzatore non sarà eseguito (a meno di chiamare successivamente `finalize_again`).

Siccome il codice dei finalizzatori non deve rispettare alcuna condizione particolare, l'implementazione di rust-cc deve verificare che il loro uso non porti alla liberazione di memoria ancora in uso. Essendo tale analisi dispendiosa, è possibile disabilitare[1] la feature `finalization` per evitare che vengano chiamati i finalizzatori.

Tuttavia, nonostante si possa disabilitare la feature `finalization`, implementare il finalizzatore è sempre obbligatorio (per alcuni motivi di compatibilità tra crate approfonditi nell'appendice A). Infatti, il trait `Finalize` è supertrait di `Trace`, come riportato nel codice 3.2.

Anche se è sempre richiesto implementare il trait `Finalize`, si noti che il metodo `finalize` non verrà mai chiamato nel caso la finalizzazione non venga mai abilitata nell'albero delle dipendenze.

[1] La feature `finalization` è abilitata di default.

Perciò, nel caso il proprio codice non sfrutti la finalizzazione è consigliabile implementare un finalizzatore vuoto oppure utilizzare l'implementazione di default del metodo `finalize`, che è vuota.

3.4.1 Macro derivativa `Finalize`

Siccome non è comune dover implementare un finalizzatore che non sia vuoto, rust-cc provvede la macro derivativa `#[derive(Finalize)]`, che semplicemente implementa il trait `Finalize` per il tipo annotato utilizzando l'implementazione vuota del metodo `Finalize::finalize`.

Un esempio di utilizzo della macro è mostrato nel codice 3.7.

```
#[derive(Finalize)]
struct Foo {
    ...
}
```

Codice 3.7: Esempio di utilizzo della macro derivativa `Finalize`.

3.5 Esempio di creazione di un ciclo di referenze

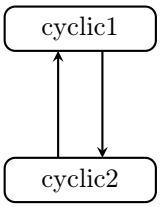
Il metodo più semplice per creare un ciclo di referenze è creare un nuovo tipo ed aggiungere un campo di tipo `RefCell<Option<Cc<Self>>>`, come mostrato nel codice 3.8.

```
#[derive(Trace, Finalize)]
struct Cyclic {
    cyclic: RefCell<Option<Cc<Self>>>,
}
```

Codice 3.8: Esempio di tipo che permette un ciclo di referenze.

La `RefCell` può essere quindi utilizzata per creare un ciclo di referenze, come mostrato nel codice 3.9.

```
let cyclic1 = Cc::new(Cyclic {
    cyclic: RefCell::new(None),
});
let cyclic2 = Cc::new(Cyclic {
    cyclic: RefCell::new(None),
});
*cyclic1.cyclic.borrow_mut() = Some(cyclic2.clone());
*cyclic2.cyclic.borrow_mut() = Some(cyclic1.clone());
```



Codice 3.9: Esempio di creazione di un ciclo di referenze.

3.6 Weak pointer

I weak pointer sono implementati tramite lo smart pointer `Weak<T>` che, a differenza di `Cc<T>`, non possiede l'allocazione puntata e non ne impedisce il collezionamento.

`Weak` non è direttamente dereferenziabile (non implementa il trait `Deref`), in quanto l'allocazione puntata potrebbe essere già stata liberata, ed è quindi necessario prima eseguirne l'upgrade ad uno strong pointer tramite il metodo `Weak::upgrade`. Tale metodo ritorna un `Option<Cc<T>>`, che conterrà un valore solo se l'allocazione puntata è ancora accessibile.

Per ottenere un nuovo weak pointer è invece necessario eseguire il downgrade a partire da un `Cc`. Tuttavia, non tutte le allocazioni supportano la creazione di weak pointer, ma solo quelle di tipo `Weakable`. Perciò, solo il tipo `Cc<Weakable<T>>` permette il downgrade ad un weak pointer, di tipo `Weak<T>`, utilizzando il metodo `downgrade`.

Per facilitare l'utilizzo di `Cc<Weakable<T>>` è presente l'alias `WeakableCc<T>`, la cui definizione è riportata nel codice 3.10.

```rust
pub type WeakableCc<T> = Cc<Weakable<T>>;
```

Codice 3.10: Definizione dell'alias `WeakableCc`.

Per creare un nuovo `WeakableCc` è possibile utilizzare le due funzioni associate `new_weakable` e `new_cyclic`, di cui è possibile vedere un esempio di utilizzo nei codici 3.11 e 3.12.

In particolare, `new_cyclic` permette la creazione di un `WeakableCc` avendo accesso ad un weak pointer ad esso, accettando una chiusura che ha come unico parametro un `&Weak` e che ritorna una istanza del tipo da allocare[2], aiutando quindi nella creazione di strutture cicliche.

Infine, è importante notare che `Weakable<T>` implementa `Deref`, essendo quindi dereferenziabile nel tipo `T` contenuto, che `Weak` può essere clonato (allo stesso modo di `Cc`) e che è possibile utilizzare il metodo `weak_count` per ottenere il numero totale di puntatori `Weak` all'allocazione puntata.

```rust
let weakable: WeakableCc<i32> = Cc::new_weakable(5);
let weak_ptr = weakable.downgrade();
let upgraded = weak_ptr.upgrade();

// weakable, weak_ptr e upgraded puntano tutti
// alla stessa allocazione, che è accessibile
assert!(upgraded.is_some());
assert_eq!(**upgraded.unwrap(), **weakable);

// Dopo aver distrutto ogni Cc (upgraded è distrutto dalla chiamata
// ad unwrap), non è più possibile eseguire l'upgrade di weak_ptr
drop(weakable);
assert!(weak_ptr.upgrade().is_none());
```

Codice 3.11: Esempio di utilizzo dei weak pointer.

[2] Ovviamente, tentare di eseguire l'upgrade di tale weak pointer durante l'esecuzione della chiusura ritornerà `None`, in quanto l'allocazione non è ancora stata inizializzata.

```rust
#[derive(Trace, Finalize)]
struct Cyclic {
    cyclic: Weak<Self>,
}

let cyclic = Cc::new_cyclic(|weak| {
    Cyclic {
        cyclic: weak.clone(),
    }
});
```

Codice 3.12: Esempio di utilizzo di `new_cyclic`.

3.7 Cleaner

Come anticipato nel paragrafo 2.4, i cleaner forniscono un'alternativa al meccanismo della finalizzazione. Una volta creata una istanza del tipo `Cleaner`, è possibile registrare delle chiusure, chiamate pulitori, che verranno eseguite durante l'esecuzione del distruttore del `Cleaner` nel quale erano state registrate.

Per creare un `Cleaner` è possibile utilizzare la funzione associata `Cleaner::new`, mentre il metodo `register` permette la registrazione di un pulitore.

Inoltre, tale metodo ritorna una istanza di `Cleanable`, che presenta il metodo `clean` per eseguire anticipatamente il pulitore registrato. Il pulitore è eseguito al massimo una volta; chiamare il metodo `clean` successivamente alla sua esecuzione (o multiple volte) non avrà alcun effetto.

È importante notare che i valori catturati dai pulitori non verranno liberati almeno fino all'esecuzione del pulitore stesso, perciò registrare un pulitore che ha accesso all'allocazione contenente il `Cleaner` porterà ad un memory leak ed è, quindi, da evitare.

Un esempio di utilizzo dei cleaner è presente nel codice 3.13.

```rust
#[derive(Trace, Finalize)]
struct Foo {
    cleaner: Cleaner,
    ...
}

let foo = Cc::new(Foo {
    cleaner: Cleaner::new(),
    ...
});

let cleanable = foo.cleaner.register(move || {
    // Codice del pulitore
});

// Per chiamare il pulitore manualmente
cleanable.clean();
```

Codice 3.13: Esempio di utilizzo dei cleaner.

3.8 State

Il modulo `state` permette di ottenere informazioni riguardo allo stato del cycle collector attraverso alcune funzioni come:

- `allocated_bytes`, che ritorna il numero di byte allocati che sono gestiti da rust-cc;
- `executions_count`, che fornisce il numero di esecuzioni del collezionamento dei cicli;
- `is_tracing`, che restituisce vero se e solo se sta venendo effettuato un tracciamento (vedi 3.3).

3.9 Collezionamento automatico dei cicli

Se la feature `auto-collect` è abilitata, il collezionamento dei cicli ha la possibilità di venire effettuato automaticamente ogni volta che viene creata un nuova allocazione. Il collezionamento può sempre essere chiamato manualmente utilizzando la funzione `collect_cycles()`.

I criteri per l'esecuzione automatica sono spiegati in dettaglio nel paragrafo 5.2.2.

3.9.1 Config

Tutti i parametri relativi al collezionamento automatico sono modificabili tramite il singleton `Config`, accessibile tramite la funzione `config`, la cui segnatura è mostrata nel codice 3.14.

```
pub fn config<F, R>(f: F) -> Result<R, ConfigAccessError>
where
    F: FnOnce(&mut Config) -> R;
```

Codice 3.14: Segnatura della funzione `config`.

Inoltre, le impostazioni non sono condivise, ma locali di ogni thread.

3.10 Supporto a target no-std

rust-cc offre supporto ad alcuni target senza libreria standard, comunemente chiamati target no-std, come i dispositivi embedded.

La feature `std`, quando non attivata[3], non richiede la presenza della libreria standard per la compilazione di rust-cc e permette quindi l'uso del cycle collector in ambienti no-std.

[3] La feature `std` è abilitata di default.

Tuttavia, siccome rust-cc necessita dell'uso di variabili locali ai thread e la macro `thread_local!{...}` è presente solamente nella standard library, è sempre necessario attivare anche la feature `nightly` per permettere l'uso dell'attributo (non ancora stabile) `#[thread_local]` al suo posto.

Questo significa anche che tutti i target supportati devono anche supportare gli ELF Thread Local Storage (ELF TLS), in quanto richiesti da tale attributo.

Capitolo 4

Algoritmo di collezionamento

Nel caso siano presenti cicli di referenze, essi vengono individuati e collezionati dall'algoritmo di collezionamento. L'algoritmo è simile quelli presentati da Lins [10] e da Bacon e Rajan [5], ma differisce per essere resistente ai panic durante il tracciamento, permettere la finalizzazione degli oggetti, non utilizzare memoria heap supplementare durante il collezionamento e supportare target senza libreria standard.

Questo capitolo tratterà in maniera astratta l'algoritmo di collezionamento dei cicli e il meccanismo della finalizzazione, mentre i dettagli implementativi saranno discussi nel prossimo capitolo.

Inoltre, nel resto del capitolo il termine "oggetto" verrà utilizzato per riferirsi, in maniera astratta, ai valori gestiti dal cycle collector.

4.1 Layout degli oggetti allocati

Ogni oggetto presenta un header, come mostrato in figura 4.1, contenente il reference counter e i campi necessari per eseguire la collezione dei cicli.

L'header contiene i campi `Next` e `Prev`, che permettono di mantenere l'oggetto in una lista intrusiva. Esistono tre liste: la prima, `POSSIBLE_CYCLES`, viene mantenuta per tutta la durata dell'esecuzione del programma e contiene gli oggetti che potrebbero fare parte di un ciclo (ovvero gli oggetti da cui parte l'identificazione dei cicli). Le

Figura 4.1: Layout di un oggetto allocato di tipo `T`.

altre due, `non_root_list` e `root_list`, vengono utilizzate solo durante la collezione dei cicli per tenere traccia di quali oggetti sono stati già tracciati in ogni momento del collezionamento.

L'header contiene anche il puntatore alla vtable del tipo T, approfondito nel paragrafo 5.1.2, che è necessario per poter tracciare, finalizzare e chiamare il distruttore di T quando si ha solo un puntatore all'header dell'oggetto e quindi non si conosce precisamente il tipo T dell'oggetto allocato.

Infine, lo stato e il reference counter dell'oggetto viene salvato all'interno del campo `CounterMarker`, mostrato in figura 4.2.

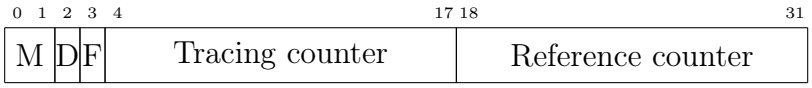


Figura 4.2: Layout di `CounterMarker`

In particolare, `CounterMarker` mantiene le seguenti informazioni:

- **M**: lo stato di marcatura dell'oggetto. Può assumere 3 tre valori:
  - `NON_MARKED` (*000*): l'oggetto non è marcato e non è contenuto in nessuna lista;
  - `IN_POSSIBLE_CYCLES` (*001*): l'oggetto non è marcato, ma potrebbe essere parte di un ciclo ed è quindi contenuto nella lista `POSSIBLE_CYCLES`;

  - `TRACED` (*010*): l'oggetto è marcato in quanto tracciato durante il collezionamento ed è contenuto in `non_root_list` oppure in `root_list`;

- **D**: lo stato di deinizializzazione. `0` se il distruttore dell'oggetto non è ancora stato eseguito, `1` altrimenti;
- **F**: lo stato di finalizzazione. `0` se l'oggetto non è ancora stato finalizzato, `1` altrimenti;
- **Tracing counter**: il contatore degli strong pointer all'oggetto tracciati durante il collezionamento;
- **Reference counter**: il contatore degli strong pointer totali all'oggetto.

4.2 Collezionamento in assenza di cicli

L'algoritmo di collezione in assenza di cicli di referenze si comporta come un algoritmo di reference counting senza collezione dei cicli spazzatura: appena il reference counter raggiunge lo zero, l'oggetto allocato viene finalizzato e, se il reference counter rimane a zero dopo la fase di finalizzazione, viene deallocato.

4.3 Collezionamento in presenza di cicli

Per individuare i cicli spazzatura, ovvero i cicli tali che tutti i puntatori ad esso sono interni al ciclo, ogni volta che uno strong pointer viene distrutto ed il suo reference counter è maggiore di zero, l'oggetto puntato viene aggiunto alla lista `POSSIBLE_CYCLES`. Come anticipato precedentemente, tale lista contiene gli oggetti che hanno la possibilità di far parte di un ciclo spazzatura e funge da buffer degli oggetti da analizzare per identificarli. Infatti, la prima fase di tracciamento partirà proprio dagli oggetti in `POSSIBLE_CYCLES` per cercare i cicli spazzatura.

Aggiungere gli oggetti alla lista `POSSIBLE_CYCLES` invece che eseguire ogni volta l'algoritmo di collezionamento è importante per evitare inutili esecuzioni del collezionamento, che porterebbero ad un degrado delle performance del cycle collector (vedi 5.2.2 e l'esempio in figura 4.3).

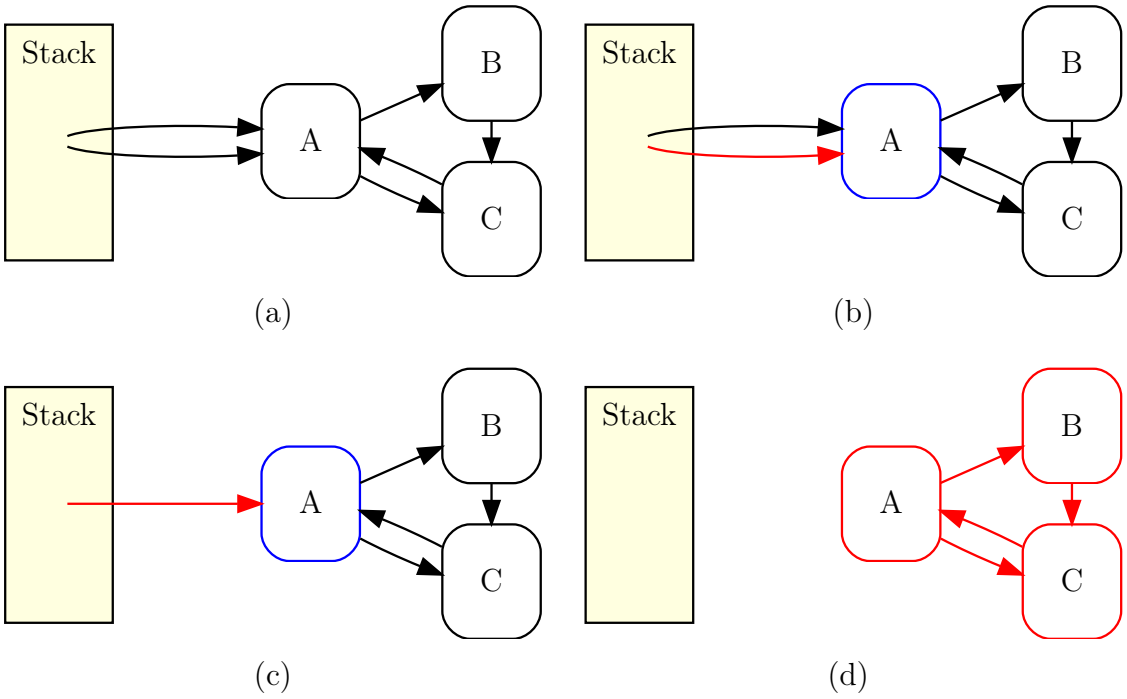


Figura 4.3: Esempio di collezionamento di un ciclo di referenze.
Il ciclo è originariamente vivo in quanto esistono 2 referenze dallo stack all'oggetto A (figura a). Successivamente, uno dei puntatori viene distrutto (in rosso, figura b) e l'oggetto A viene aggiunto a `POSSIBLE_CYCLES` (in blu). Se il collezionamento fosse eseguito adesso, nessun oggetto sarebbe collezionato. Dopo che anche l'ultima referenza dallo stack verrà distrutta (figura c) e la collezione sarà eseguita, tutti gli oggetti del ciclo verranno collezionati (figura d).

4.3.1 Individuamento dei cicli spazzatura

Per individuare i cicli spazzatura vengono effettuate due fasi di tracciamento. Durante le due fasi vengono utilizzate le due liste `non_root_list` e `root_list` per mantenere traccia degli oggetti tracciati e dei cicli (spazzatura) scoperti.

Trace counting

L'obbiettivo della prima fase, chiamata trace counting, è di contare il numero degli strong pointer ad ogni oggetto raggiungibili a partire dagli oggetti nella lista `POSSIBLE_CYCLES`. Viene quindi effettuata una visita in profondità a partire da ogni elemento della lista e marcando ogni oggetto visitato come `TRACED`. Ogni volta che un

puntatore viene seguito, il tracing counter[1] dell'oggetto puntato viene incrementato di uno. Inoltre, ogni volta che il tracing counter è incrementato viene comparato con il reference counter: se sono identici allora l'oggetto verrà inserito (o spostato) nella lista `non_root_list`, altrimenti verrà inserito (o spostato) nella lista `root_list`.

Questo primo tracciamento permette quindi sia di scoprire il sottografo (del grafo degli oggetti allocati nello heap) formato dagli oggetti raggiungibili a partire da `POSSIBLE_CYCLES`, sia di conoscere il numero di puntatori appartenenti al sottografo che puntano ad ogni elemento del sottografo stesso. Il sottografo sarà composto dall'unione degli oggetti delle due liste `non_root_list` e `root_list`.

Inoltre, comparare il tracing counter ed il reference counter permette di identificare correttamente le *radici* del sottografo, ovvero gli oggetti appartenenti al sottografo individuato sicuramente vivi e non spazzatura: se il tracing counter di un oggetto è strettamente minore del suo reference counter alla fine del tracciamento, allora deve esistere almeno un puntatore esterno al sottografo che punta ad esso.

Le radici verranno usate nella seconda fase di tracciamento per identificare gli oggetti appartenenti al sottografo che non sono spazzatura.

Alla fine della fase di trace counting, le radici saranno contenute all'interno della lista `root_list`, mentre i restanti oggetti appartenenti al sottografo saranno contenuti nella lista `non_root_list`.

Trace roots

La seconda fase di tracciamento, chiamata trace roots, è simile ad una fase di marcatura di un garbage collector mark-and-sweep: viene infatti effettuato un tracciamento a partire dalle radici individuate dalla fase di trace counting, rimuovendo tutti gli oggetti incontrati dalle rispettive liste e marcandoli come `NON_MARKED`.

[1] Questo uso di un contatore diverso dal reference counter differisce dall'approccio utilizzato da Lins[10] e da Bacon e Rajan[5], che decrementano il reference counter stesso invece di usare un contatore separato, ma è necessario in quanto sarebbe praticamente impossibile ripristinare il corretto valore del contatore nel caso di un panic durante il tracciamento, cosa che è ammessa dal contatto di `Trace` (vedi 3.3.1). L'invariante legata al tracing counter è che il suo valore deve essere tra zero ed il valore del reference counter (inclusi).

Al termine del tracciamento si avranno quindi tutti gli oggetti appartenenti a cicli spazzatura all'interno della lista `non_root_list` (mentre la lista `root_list` sarà vuota).

4.3.2 Analisi dell'identificazione dei cicli

Il caso pessimo dell'identificazione dei cicli è $O(N + P)$, dove N è il numero di oggetti allocati sullo heap e P è il numero degli strong pointer esistenti. Infatti, l'identificazione dei cicli traccerà al massimo l'intero grafo degli oggetti allocati sullo heap. Tuttavia, in caso di heap dove il grafo degli oggetti allocati è suddiviso in più sottografi non connessi tra di loro, l'algoritmo presentato presenta performance migliori di quelle di un garbage collector mark-and-sweep, in quanto verrà analizzata soltanto una parte degli oggetti allocati.

Inoltre, non viene richiesta memoria ausiliaria (ad eccezione di memoria stack) durante i tracciamenti ed è dunque possibile utilizzare l'algoritmo su dispositivi embedded o con poca memoria heap disponibile.

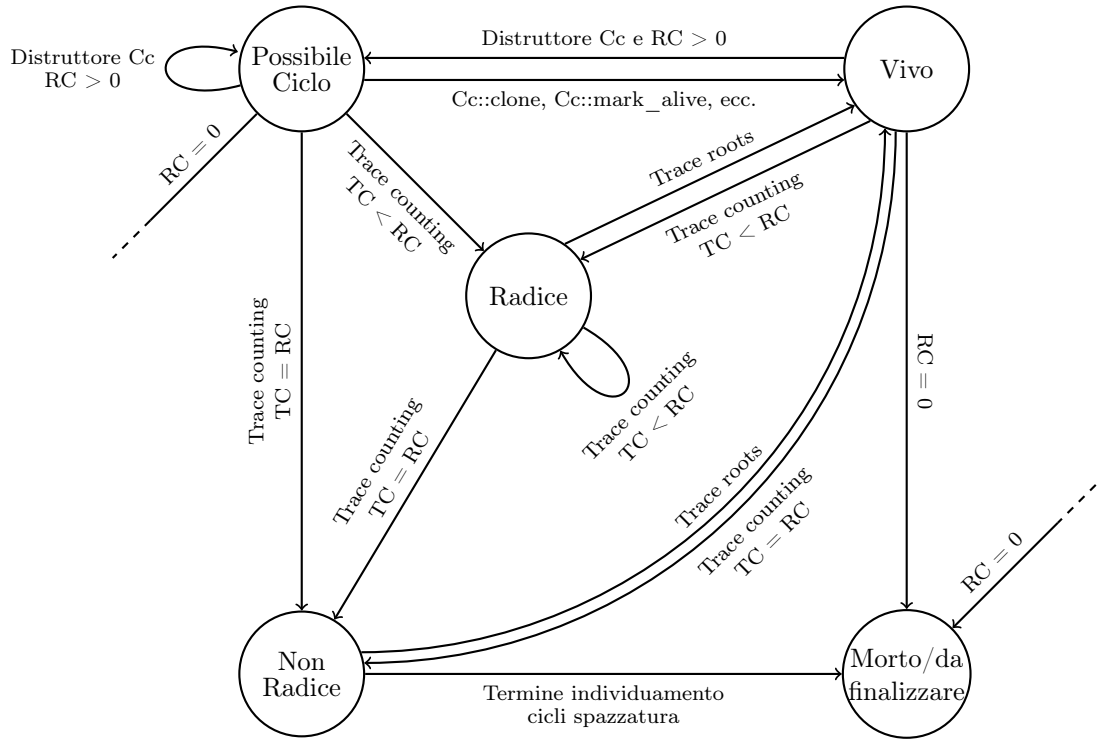
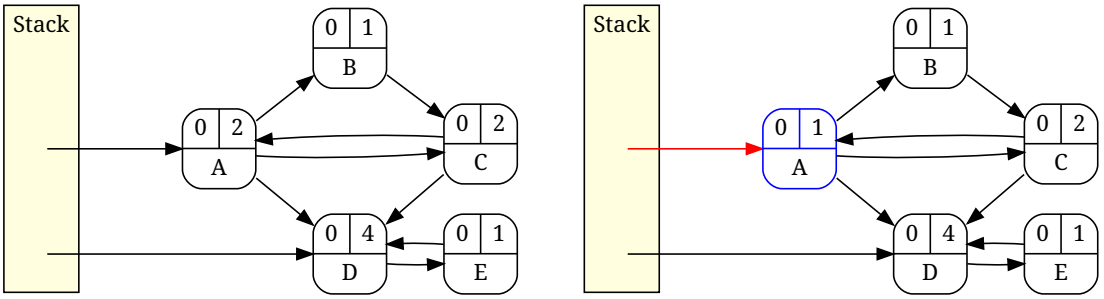


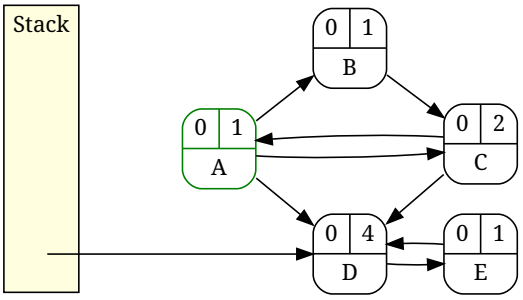
Figura 4.4: Diagramma degli stati di un oggetto.


Stack
0 1
B
0 2
A
0 2
C
0 4
D
0 1
E

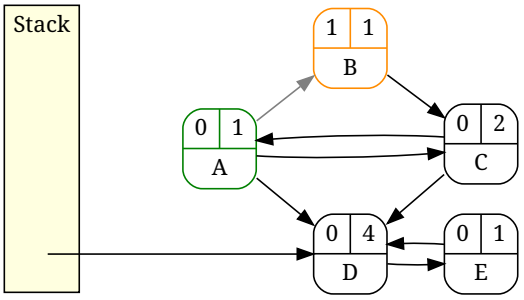
(a)

Stack
0 1
B
0 1
A
0 2
C
0 4
D
0 1
E

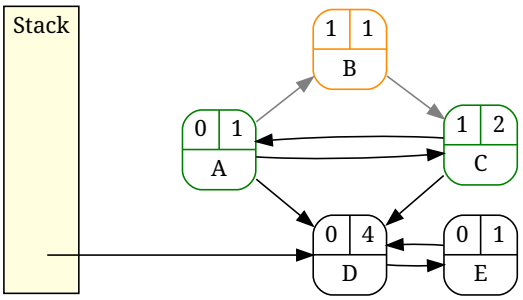
(b)


Stack
0 1
B
0 1
A
0 2
C
0 4
D
0 1
E

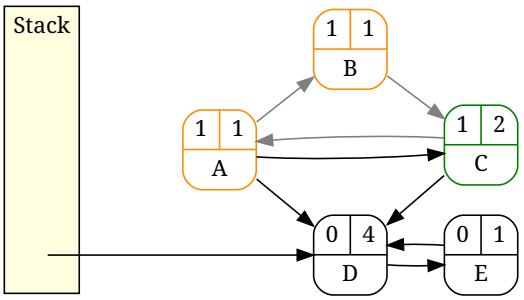
(c)


Stack
1 1
B
0 1
A
0 2
C
0 4
D
0 1
E

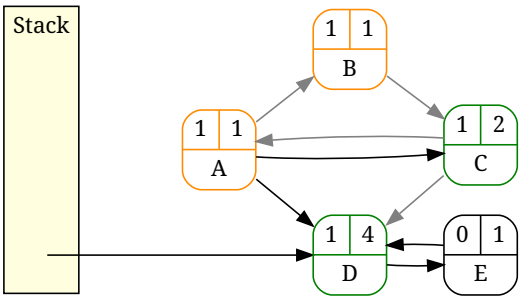
(d)


Stack
1 1
B
0 1
A
1 2
C
0 4
D
0 1
E

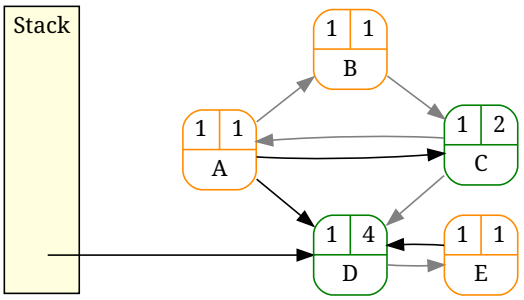
(e)


Stack
1 1
B
1 1
A
1 2
C
0 4
D
0 1
E

(f)


Stack
1 1
B
1 1
A
1 2
C
1 4
D
0 1
E

(g)


Stack
1 1
B
1 1
A
1 2
C
1 4
D
1 1
E

(h)

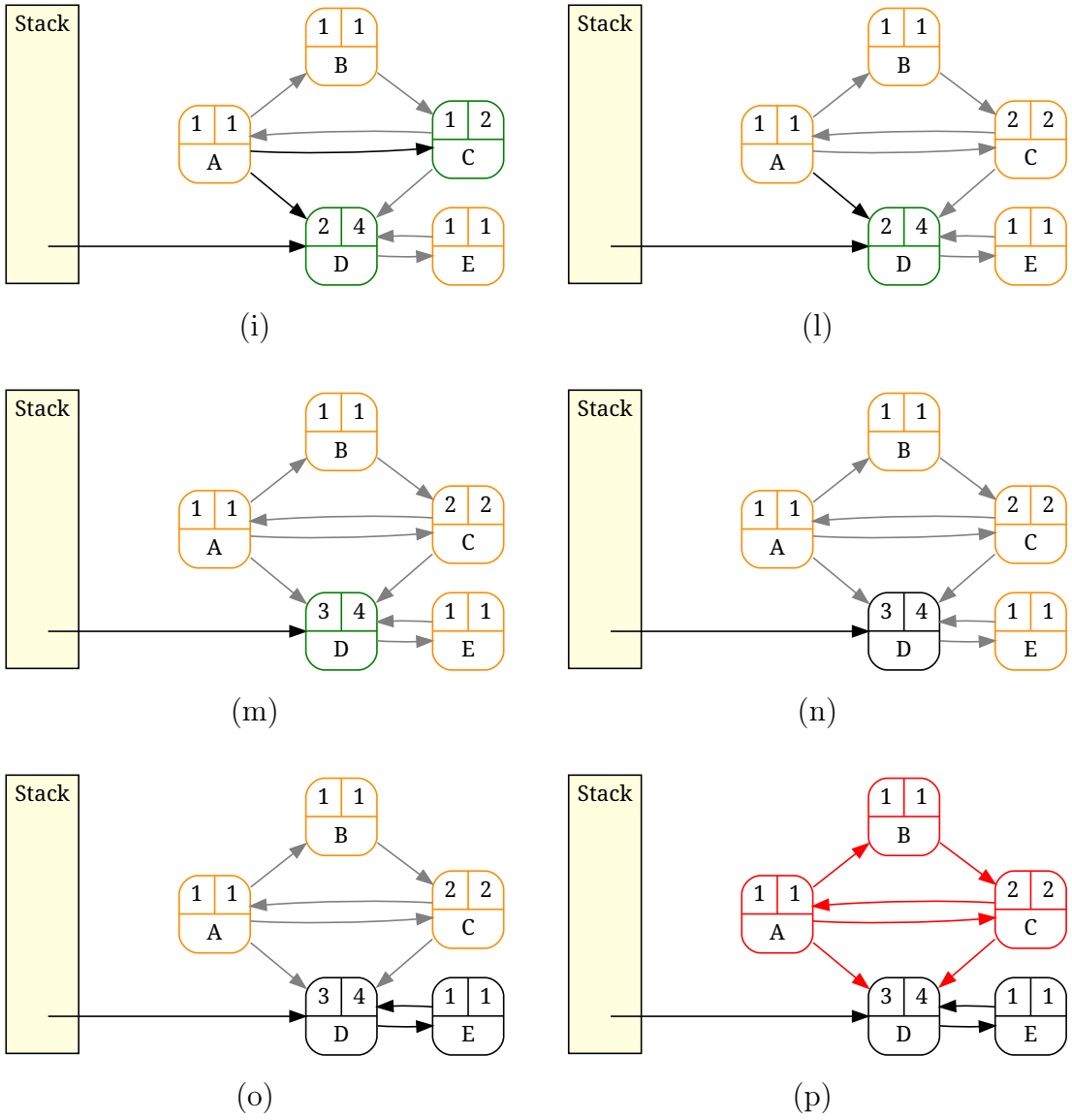


Figura 4.5: Esempio di esecuzione dell'algoritmo di collezionamento.
Il ciclo è inizialmente vivo, con una referenza dallo stack verso l'oggetto A ed una verso l'oggetto D (figura a). Il reference counter di ogni oggetto è riportato in alto a destra, mentre il tracing counter è in alto a sinistra. Successivamente, la referenza verso l'oggetto A viene distrutta ed A viene inserito nella lista `POSSIBLE_CYCLES` (in blu, figura b). Inizia quindi la fase di trace counting ed A viene inserito in `root_list` (in verde, figura c). Continua quindi il tracciamento fino a tracciare ogni oggetto raggiungibile da A. I puntatori già tracciati sono in grigio, mentre gli oggetti in `non_root_list` sono in arancione (figure da d a m). Inizia la fase di trace roots: l'oggetto D è l'unica radice ed è quindi vivo (figura n). Anche E è vivo in quanto tracciato partendo da D (figura o). Nessun altro oggetto è tracciato, quindi gli oggetti A, B e C sono collezionati (figura p).

4.3.3 Proprietà di correttezza

Definizione (Correttezza)**.** L'identificazione dei cicli si dice *corretta* quando identifica soltanto cicli che sono spazzatura.

Teorema. *L'algoritmo di identificazione dei cicli è corretto.*

Dimostrazione. La fase di trace counting produce le due liste `non_root_list` e `root_list`. Insieme, esse costituiscono il sottografo (del grafo degli oggetti allocati nello heap) tracciato a partire dagli oggetti in `POSSIBLE_CYCLES`.

`non_root_list` contiene tutti gli oggetti tracciati tali che il tracing counter è uguale al reference counter, ovvero gli oggetti per i quali tutte le loro referenze sono contenute nel sottografo.

Invece, `root_list` contiene tutti gli oggetti tracciati tali che il tracing counter è minore del reference counter, ovvero gli oggetti per i quali esistono referenze esterne al sottografo.

Durante la seconda fase vengono rimossi dalla lista `non_root_list` tutti gli oggetti che sono raggiungibili dagli oggetti in `root_list`. Quindi, al termine della fase di trace roots, `non_root_list` formerà un secondo sottografo contenente solamente oggetti tali che tutte le loro referenze sono contenute in questo sottografo, ovvero i cicli spazzatura.

Perciò, l'algoritmo presentato identifica soltanto cicli che sono spazzatura ed è quindi corretto. □

4.3.4 Proprietà di completezza

Definizione (Completezza)**.** L'identificazione dei cicli si dice *completa* quando identifica tutti i cicli spazzatura esistenti al momento dell'esecuzione.

Teorema. *L'algoritmo di identificazione dei cicli è completo.*

Dimostrazione. Il sottografo (del grafo degli oggetti allocati nello heap) tracciato durante la fase di trace counting contiene tutti i cicli spazzatura esistenti al momento dell'esecuzione del tracciamento poiché, considerato un ciclo, per diventare

spazzatura esso deve essere non più raggiungibile e quindi tutte le referenze ai membri del ciclo, esterne al ciclo stesso, devono essere state distrutte. Siccome l'ultima di tali referenze distrutte sarà sicuramente contenuta in `POSSIBLE_CYCLES` (all'inizio del tracciamento successivo all'esecuzione del suo distruttore), il ciclo verrà tracciato durante questa prima fase.

La seconda fase, trace roots, esclude da tale sottografo tutti e soli gli oggetti ancora vivi, siccome `root_list` contiene tutti gli oggetti per i quali esistono referenze non appartenenti al sottografo (vedi proprietà di correttezza) e tutti gli oggetti raggiunti a partire dagli oggetti in tale lista sono considerati vivi.

Quindi, i rimanenti oggetti nel sottografo devono essere tutti gli oggetti appartenenti a cicli spazzatura, in quanto sono i soli oggetti tali che tutte le referenze a loro sono interne al sottografo stesso.

Perciò, l'algoritmo presentato identifica tutti i cicli spazzatura esistenti al momento dell'esecuzione ed è quindi completo. □

4.4 Finalizzazione

L'esecuzione dei finalizzatori può avvenire in due momenti:

1. quando il reference counter di un oggetto raggiunge 0;
2. subito dopo l'identificazione dei cicli e prima che vengano liberati.

Siccome è possibile che alcuni oggetti siano stati resuscitati durante la finalizzazione, prima di procedere con il liberamento della memoria è necessario controllare che gli oggetti non risultino di nuovo accessibili da parte del programma.

Riconsiderando i due casi in cui può avvenire la finalizzazione, sono necessarie le seguenti misure:

1. nel caso il reference counter abbia raggiunto 0, basta verificare che il reference counter sia rimasto a 0 successivamente alla finalizzazione. In tal caso, l'oggetto non è stato resuscitato e quindi si può procedere al liberamento del-

la memoria, altrimenti si inserirà l'oggetto nella lista `POSSIBLE_CYCLES`, in quanto è possibile che ora sia contenuto all'interno di un ciclo;

2. nel caso la finalizzazione venga effettuata durante la collezione dei cicli, invece, è necessario eseguire nuovamente l'identificazione dei cicli (successivamente alla finalizzazione). Gli oggetti verranno dunque inseriti nuovamente all'interno di `POSSIBLE_CYCLES` e la procedura di collezionamento ripartirà dall'inizio.

Capitolo 5

Implementazione

In questo capitolo verranno trattati gli aspetti specifici dell'implementazione di rust-cc. In particolare, come vengono allocati i valori sullo heap, il funzionamento dello smart pointer `Cc` e i dettagli implementativi dell'algoritmo di collezionamento, dei weak pointers e dei cleaners.

5.1 Smart pointer `Cc` e il tipo `CcBox`

Lo smart pointer `Cc` è definito come mostrato nel codice 5.1 e punta ad una istanza della struct `CcBox` allocata sullo heap. Il parametro generico `T` presenta come bound il trait `Trace` per permettere il tracciamento del valore allocato, mentre il campo di tipo `PhantomData<Rc<T>>` serve ad evitare che `Cc` implementi i trait `Send` e `Sync`, siccome il collector è single threaded.

```
#[repr(transparent)]
pub struct Cc<T: ?Sized + Trace + 'static> {
    inner: NonNull<CcBox<T>>,
    _phantom: PhantomData<Rc<T>>,
}
```

Codice 5.1: Definizione dello smart pointer `Cc`.

La definizione della struct `CcBox` è presente nel codice 5.2. Come anticipato nel paragrafo 4.1, è costituita da un header, formato dai campi `next`, `prev`, `vtable`, `fat_ptr` e `counter_marker`, e dal valore allocato nel campo `elem`.

```
#[repr(C)]
pub(crate) struct CcBox<T: ?Sized + Trace + 'static> {
    next: UnsafeCell<Option<NonNull<CcBox<()>>>>,
    prev: UnsafeCell<Option<NonNull<CcBox<()>>>>,

    #[cfg(feature = "nightly")]
    vtable: DynMetadata<dyn InternalTrace>,

    #[cfg(not(feature = "nightly"))]
    fat_ptr: NonNull<dyn InternalTrace>,

    counter_marker: CounterMarker,

    elem: UnsafeCell<T>,
}
```

Codice 5.2: Definizione della struct `CcBox`.

5.1.1 Implementazione delle liste intrusive

I campi `next` e `prev` permettono di mantenere i `CcBox<T>` in una lista intrusiva.

Si noti che il tipo puntato da tali campi è `CcBox<()>` e non `CcBox<U>` (per un qualche tipo `U`), siccome non è possibile sapere a tempo di compilazione quale sarà il tipo del parametro generico `T` degli altri `CcBox` nella lista.

Le liste intrusive, quindi, permettono l'accesso solamente agli header e non al valore allocato, il cui tipo è posto come unit dal momento che, avendo dimensione zero ed allineamento uno, non permette l'accesso al valore contenuto a meno dell'utilizzo di cast espliciti.

5.1.2 Puntatore alla vtable

L'header contiene anche il puntatore alla vtable del tipo T. Tale puntatore è necessario per poter tracciare, finalizzare e chiamare il distruttore di T quando si ha solo un puntatore all'header, ad esempio durante la scansione di una lista intrusiva.

Per implementare tale puntatore viene utilizzato il trait object `dyn InternalTrace`. `InternalTrace` è un trait privato, implementato solamente per `CcBox`, che serve a generare la vtable del valore allocato.

Per salvare tale puntatore, viene usato il tipo `DynMetadata<dyn InternalTrace>` della standard library sulla toolchain NIGHTLY, mentre su STABLE viene utilizzato un fat pointer autoreferenziale `NonNull<dyn InternalTrace>` che punta allo stesso `CcBox` nel quale è salvato. Utilizzare un fat pointer incrementa la dimensione dell'header di una parola, ma è necessario in quanto `DynMetadata` non è ancora stabilizzato ed è quindi presente solamente sulla toolchain NIGHTLY.

5.1.3 CounterMarker

Lo stato e il reference counter sono salvati all'interno del bit field `counter_marker`.

L'implementazione del bit field è effettuata dal tipo `CounterMarker`, definito come una `Cell<u32>`, ed è tale che rispetta il layout mostrato nel codice 4.2.

5.1.4 Gestione dell'aliasing dei puntatori

Molte operazioni, come clonare uno smart pointer `Cc` oppure chiamare il distruttore di un valore, richiedono di modificare il contenuto dei campi salvati nei `CcBox` anche in presenza di referenze immutabili. Viene quindi impiegato il pattern dell'interior mutability, tramite l'uso di `UnsafeCell` e `Cell`, per ottenere referenze mutabili a partire da referenze immutabili e permettere quindi la modifica del contenuto dei campi.

5.2 Implementazione del collezionamento

5.2.1 Identificazione dei cicli

L'implementazione dell'identificazione dei cicli segue l'algoritmo mostrato nel capitolo 4.

In particolare, il distruttore di `Cc` decrementa il reference counter e inserisce il `CcBox` nella lista `POSSIBLE_CYCLES` se il reference counter rimane maggiore di zero. La lista `POSSIBLE_CYCLES` è salvata in una variabile thread local ed la fase di trace counting viene effettuata chiamando il metodo `trace` su ogni elemento di tale lista. Analogamente, la fase di trace roots viene effettuata chiamando il metodo `trace` su ogni elemento della lista `root_list`.

Inoltre, l'aggiunta e rimozione dei `CcBox` dalle liste `root_list` e `non_root_list` è gestita dall'implementazione del metodo `Trace::trace` di `Cc`. Per poter accedere a tali liste, esse sono passate per referenza mutabile all'interno della struct `Context` (vedi descrizione del codice 3.2).

5.2.2 Attivazione del collezionamento automatico

Come mostrato in precedenza nel paragrafo 3.9, il collezionamento ha la possibilità di avvenire ogni volta che viene creata una nuova allocazione oppure chiamando manualmente la funzione `collect_cycles()`. Si noti, inoltre, che un collezionamento non può mai avvenire durante l'esecuzione di un altro collezionamento.

Per evitare che le performance del collector degradino in maniera quadratica col numero di allocazioni, si deve evitare che il collezionamento avvenga un numero di volte pari al numero di allocazioni. A tale scopo, il collezionamento non avviene ogni volta che viene creata una nuova allocazione, ma soltanto quando il numero di byte allocati supera un certo limite (threshold). Tale limite di default è impostato a 100 byte.

Se dopo un collezionamento tale limite è inferiore al numero di byte allocati, allora il limite viene raddoppiato finché non supera i byte allocati. Questo assicura che il collezionamento non avvenga immediatamente alla prossima allocazione.

Questo approccio evita quindi di eseguire una collezione per ogni allocazione, tuttavia non evita che, dopo grandi deallocazioni, il prossimo collezionamento sia molto posticipato nel tempo, in quanto il limite è rimasto molto alto rispetto ai byte correntemente allocati. Per risolvere quest'ultimo problema, si considera una percentuale di aggiustamento, impostata di default a 0.1, tale che se il limite moltiplicato per tale percentuale è maggiore rispetto al numero di byte allocati, allora il limite viene dimezzato finché la condizione non risulti vera.

Nell'eseguire tali dimezzamenti è comunque necessario evitare che il limite finale risulti inferiore ai byte allocati, perciò se successivamente al dimezzamento il limite rompesse tale invariante, allora il threshold finale sarà quello prima della divisione.

In caso si desiderasse disabilitare la percentuale di aggiustamento in particolari applicazioni, è sufficiente impostare tale percentuale a zero. Impostare la percentuale a 1, invece, fa diminuire dopo ogni collezione il limite al multiplo di 100 (il threshold iniziale) successivo dopo il numero di byte allocati.

Infine, esiste un altro caso in cui il collezionamento può avvenire quando viene creata una nuova allocazione, che tuttavia è disabilitato di default, siccome non è ideale per la maggior parte delle applicazioni. È possibile, infatti, impostare un limite al numero di elementi nella lista `POSSIBLE_CYCLES` (vedi 4.3). Se il numero di elementi superasse tale limite, verrà eseguita una collezione.

Come accennato precedentemente, questo approccio non è sempre ideale, ad esempio nel caso venissero frequentemente distrutti molti puntatori `Cc` ad un ciclo non spazzatura. Ciò causerebbe ripetuti (ed inutili) collezionamenti che rallenterebbero solamente l'applicazione.
Inoltre, eseguendo i benchmark discussi nel paragrafo 6.3 impostando il limite a 255, è stato possibile notare che le performance sono sempre peggiorate rispetto a quando tale limite era disabilitato.

È possibile, tuttavia, che alcune applicazioni traggano beneficio dall'applicare tale limite, che perciò è abilitabile opzionalmente impostandolo ad un valore maggiore di zero.

5.2.3 Note sulle proprietà di correttezza e completezza

È importante notare che, durante la dimostrazione delle proprietà di correttezza e completezza (vedi 4.3.3 e 4.3.4), è stata fatta l'assunzione che il sottografo (del grafo dello heap) tracciato durante la fase di trace counting non cambi tra la prima e la seconda fase del tracciamento.

Tuttavia, le implementazioni delle funzioni `trace` e codice concorrente hanno la possibilità di modificare il contenuto delle allocazioni durante il tracciamento e di conseguenza hanno la possibilità di alterare il sottografo tra tali fasi. Dunque, l'identificazione dei cicli non risulterebbe corretta, in quanto il sottografo analizzato durante la fase di trace roots potrebbe non corrispondere a quello analizzato durante quella di trace counting.

Proprio per tale ragione il trait `Trace` è unsafe da implementare e sono presenti i requisiti di safety mostrati nel paragrafo 3.3.1, che impediscono, durante la seconda fase, il tracciamento di puntatori `Cc` non tracciati durante la prima[1].

Infine, si noti che i requisiti di safety di `Trace` permettono anche di non tracciare alcuni puntatori: ogni `CcBox` puntato da un puntatore *esplicitamente*[2] escluso dal tracciamento è considerato come vivo e non verrà collezionato.

5.3 Implementazione della finalizzazione

Come anticipato nel paragrafo 4.4, la finalizzazione può avvenire in due casi:

1. durante l'esecuzione del distruttore di `Cc`, quando il reference counter raggiunge 0;
2. durante l'esecuzione dell'algoritmo di collezione dei cicli, prima che i cicli spazzatura vengano liberati.

[1] Si noti anche che tali requisiti proibiscono il tracciamento di strong pointer mutabili da altri thread, ad esempio se contenuti in un mutex. Infatti, le strutture dati per la concorrenza, come `Mutex` e `RwLock`, non implementano il trait `Trace`. [2] Ad esempio, tramite l'utilizzo dell'attributo `#[rust_cc(ignore)]`, vedi 3.3.2.

Nel primo caso, siccome il reference counter ha raggiunto 0, è importante che sia incrementato di nuovo ad 1 per la durata dell'esecuzione del finalizzatore[3]. Questo permette di rendere sound ottenere un puntatore `Cc` al `CcBox` che sta venendo finalizzato, come mostrato nel codice 5.3 facendo uso di un weak pointer all'allocazione stessa.

```
#[derive(Trace)]
struct Cyclic {
    weak: Weak<Cyclic>,
}

impl Finalize for Cyclic {
    fn finalize(&self) {
        assert_eq!(1, self.weak.strong_count());
        let this: WeakableCc<Cyclic> = self.weak.upgrade().unwrap();
        // Il CcBox puntato da this è lo stesso puntato da self
        assert!(ptr::eq(self, &**this));
        assert_eq!(2, this.strong_count());
    }
}

let _ = Cc::new_cyclic(|weak| Cyclic {
    weak: weak.clone(),
});
```

Codice 5.3: Esempio di finalizzazione con upgrade di un weak pointer.
Esempio di finalizzazione di un'allocazione (in assenza di cicli) dove, durante l'esecuzione del finalizzatore, viene creato uno strong pointer all'allocazione stessa utilizzando un weak pointer.

Il codice dell'esempio 5.3 utilizza la funzione `new_cyclic` per creare un'istanza di `WeakableCc<Cyclic>` contenente un weak pointer all'allocazione stessa. Durante la finalizzazione, tale weak pointer viene utilizzato per ottenere uno strong pointer al valore allocato.

[3] In realtà, per motivi di ottimizzazione, il reference counter non viene mai decrementato a zero prima dell'esecuzione del finalizzatore, ma viene verificato se esso sia uguale ad uno e poi successivamente decrementato appena essa termina.

Tale codice è sound e non produce comportamento non definito, in quanto il reference counter all'inizio del metodo `finalize` avrà valore 1, per poi essere incrementato a 2 durante la chiamata ad `upgrade()`. Distruggere il `Cc` ritornato da `upgrade()` decrementerà il reference counter, che tornerà ad 1, ma non libererà la memoria puntata.

Inoltre, se non si fosse incrementato il reference counter a 1 prima dell'esecuzione del finalizzatore, il distruttore[4] del `Cc` ritornato da `upgrade()` avrebbe liberato l'allocazione puntata, causando l'uso di memoria liberata e una seconda deallocazione della stessa area di memoria (causata dal distruttore che aveva originariamente iniziato la finalizzazione).

Nel secondo caso, invece, siccome gli elementi dei cicli spazzatura sono inseriti di nuovo in `POSSIBLE_CYCLES` (successivamente alla loro finalizzazione) e l'identificazione dei cicli viene eseguita nuovamente, è necessario evitare che ogni `CcBox` possa essere finalizzato un numero infinito di volte, dato che questo non farebbe terminare il collezionamento. Perciò, ogni `CcBox` è finalizzabile soltanto 1 volta (a meno che non venga chiamato il metodo `Cc::finalize_again`, che però non può essere chiamato durante il collezionamento).

In aggiunta, per evitare che casi come quelli mostrati nel codice 5.4 non facciano terminare il collezionamento (o che addirittura esauriscano la memoria stack in caso di assenza di cicli), ogni `CcBox` creato durante la fase di finalizzazione è considerato come già finalizzato. Perché venga finalizzato sarà necessario chiamare il metodo `Cc::finalize_again` in seguito al di fuori di un collezionamento.

Oltre ai casi trattati, esiste un ultimo caso patologico, trattato nell'appendice B.

Infine, siccome l'identificazione dei cicli e la finalizzazione vengono chiamate ciclicamente, è bene porre un limite superiore al numero di esecuzioni di tale ciclo, che in rust-cc è posto a 10.

[4] Notare che tale distruttore non avrebbe eseguito di nuovo il finalizzatore, dal momento che era già stato eseguito.

```rust
#[derive(Trace)]
struct ToFinalize;

impl Finalize for ToFinalize {
    fn finalize(&self) {
        // Viene creato e distrutto un nuovo CcBox (aciclico)
        // dello stesso tipo, quindi verrà eseguito nuovamente
        // questo finalizzatore che creerà un nuovo CcBox,
        // che a sua volta eseguirà questo finalizzatore, ecc.
        let _ = Cc::new(ToFinalize);
    }
}

let _ = Cc::new(ToFinalize);
```

Codice 5.4: Esempio di finalizzazione ricorsiva in assenza di cicli.

5.4 Liberamento della memoria

Se ogni `CcBox` identificato come facente parte di un ciclo spazzatura fosse già stato finalizzato, si procederà con il liberamento della memoria.

Prima di deallocare la memoria di ogni `CcBox`, è importante chiamarne il relativo distruttore. La chiamata di tutti i distruttori deve avvenire però prima della deallocazione dei `CcBox`, altrimenti i loro drop glue leggerebbero memoria liberata.

Infatti, i drop glue chiamano i distruttori dei `Cc` contenuti nel `CcBox` che sta venendo distrutto e che puntano ad altri elementi del ciclo. Tali distruttori terminano immediatamente, in quanto i `CcBox` sono marcati come `TRACED`[5], ma accedono comunque all'allocazione per leggere il `CounterMarker`.

[5] In questo caso, i distruttori dei `Cc` non devono gestire la deallocazione, in quanto sta già venendo gestita dal collezionamento. Pertanto, essi terminano immediatamente se un `CcBox` è marcato come `TRACED`.

Inoltre, è importante segnare sempre tutti i `CcBox` da deallocare come deinizializzati prima di eseguirne i distruttori. Questo è rilevante nel caso uno dei distruttori esegua un panic, in quanto questo fermerebbe l'esecuzione di tutti i restanti distruttori e avverrebbe un memory leak di tutti i cicli spazzatura.

In tale caso, rimarrebbe possibile utilizzare dei weak pointer per ottenere dei puntatori `Cc` al ciclo che stava venendo deinizializzato, e quindi sarebbe unsound.

Per evitare ciò, prima di eseguire ogni distruttore il relativo `CcBox` viene segnato come deinizializzato (vedi 4.1) e, nel caso di un panic, i restanti `CcBox` non ancora deinizializzati vengono segnati anch'essi come deinizializzati.
Quando si proverà a chiamare il metodo `Weak::upgrade` in un caso come quello appena descritto, tale metodo ritornerà `None` dal momento che il `CcBox` era stato segnato come deinizializzato.

5.5 Weak pointer

È possibile identificare due strategie principali per implementare i weak pointer:

1. mantenendo il weak counter nella stessa allocazione del `CcBox`;
2. allocando il weak counter in una nuova allocazione.

rust-cc utilizza il secondo approccio, in quanto mantenere il weak counter nella stessa allocazione significherebbe non poter liberare immediatamente la memoria dell'intero `CcBox` appena possibile, ma bisognerebbe anche aspettare che il weak counter arrivi a zero.

Il secondo approccio, invece, mantiene allocati soltanto 2 byte per ogni `CcBox`.

5.5.1 `Weak e Weakable`

Un weak pointer `Weak<T>` è formato da due puntatori: uno al `WeakMetadata`, che mantiene i metadata necessari come il weak counter, ed uno al `CcBox`.

```
pub struct Weak<T: ?Sized + Trace + 'static> {
    metadata: NonNull<WeakMetadata>,
    cc: NonNull<CcBox<Weakable<T>>>,
}
```

Codice 5.5: Definizione dello smart pointer `Weak`.

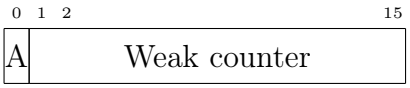


Figura 5.1: Layout di `WeakMetadata`

In particolare, `WeakMetadata` mantiene le seguenti informazioni:

- **A**: lo stato di accessibilità del `CcBox`. `1` quando è possibile dereferenziare il puntatore al `CcBox`, `0` quando l'allocazione del `CcBox` è già stata liberata;
- **Weak counter**: il contatore dei puntatori `Weak` al `CcBox`.

Per poter implementare i weak pointer, come mostrato nel paragrafo 3.6, si ha la necessità di aggiungere un wrapper `Weakable<T>`, la cui definizione è riportata nel codice 5.6[6], intorno al valore allocato. Il motivo di tale necessità è che bisogna mantenere il puntatore al `WeakMetadata` anche all'interno del `CcBox`, in maniera da poter accedere al weak counter e per segnare l'allocazione del `CcBox` come non accessibile quando viene liberata.

Inoltre, il trait `Trace` è implementato sia per `Weakable` che per `Weak`. In particolare, l'implementazione per `Weak` è vuota dal momento che, non essendo uno strong pointer, non deve tracciare il `CcBox` puntato.

[6] In realtà, la definizione di `Weakable` è leggermente più complessa, in quanto il `WeakMetadata` viene allocato soltanto durante la creazione del primo weak pointer. Questo permette di evitare allocazioni inutili nel caso non vengano mai creati weak pointer verso alcuni `CcBox`.

```rust
pub struct Weakable<T: ?Sized + Trace + 'static> {
    metadata: NonNull<WeakMetadata>,
    elem: T,
}
```

Codice 5.6: Definizione della struct `Weakable`.

5.5.2 Upgrade e downgrade di weak pointer

Mentre l'implementazione della funzione `WeakableCc::downgrade` semplicemente incrementa il weak counter e ritorna una nuova istanza di `Weak`, l'implementazione della funzione `Weak::upgrade` deve essere più cauta.

Infatti, come mostrato in 5.4, è possibile utilizzare i weak pointer per ottenere, in caso di un panic, un `Cc` ad un `CcBox` facente parte di un ciclo parzialmente deinizializzato.

Per evitare tale eventualità, `Weak::upgrade` viene definita come mostrato nel codice 5.7, e la funzione `Weak::strong_count` viene definita come segue:

1. se il `CcBox` non è accessibile, ritorna 0;
2. se il `CcBox` è segnato come deinizializzato, allora ritorna 0;
3. se il `CcBox` è marcato come tracciato e il collector è nella fase di chiamata dei distruttori, allora ritorna 0;
4. altrimenti, ritorna il numero di strong reference salvato in `CounterMarker`.

```rust
pub fn upgrade(weak: &Weak<T>) -> Option<WeakableCc<T>> {
    if weak.strong_count() == 0 {
        None
    } else {
        weak.increment_strong_count();
        return Some(weak.get_new_strong_ref());
    }
}
```

Codice 5.7: Definizione della funzione `Weak::upgrade` (in pseudo codice).

Queste condizioni bastano per evitare che si possano ottenere strong pointer a `CcBox` deinizializzati oppure facenti parte di cicli parzialmente deinizializzati. Infatti, possono avvenire 4 casi:

- il distruttore del `CcBox` è stato eseguito senza panic e quindi non risulta accessibile;
- è avvenuto un panic durante l'esecuzione del distruttore e quindi il `CcBox` è segnato come deinizializzato invece che risultare non accessibile;
- si sta accedendo ad un `CcBox` da deinizializzare, ma il cui distruttore non è ancora stato eseguito. In tal caso, il `CcBox` è marcato come tracciato ed il collector è nella fase di esecuzione dei distruttori. È importante controllare la fase del collector, in quanto anche durante la finalizzazione i `CcBox` (da finalizzare) sono marcati come tracciati, ma è permesso ottenere uno strong pointer chiamando `upgrade`;
- si sta accedendo ad un `CcBox` da deinizializzare, ma è avvenuto un panic durante l'esecuzione di un altro distruttore. In questo caso, il collector segna tutti i `CcBox` come deinizializzati, anche quelli il cui distruttore non è stato ancora eseguito (vedi 5.4), e quindi risulterà sicuramente segnato come deinizializzato.

5.6 Cleaner

I cleaner vengono implementati utilizzando i weak pointer e la struttura dati `SlotMap` del crate `slotmap`[15].

I weak pointer sono utilizzati per non dover limitare il tempo di vita dei `Cleanable` ritornati da `Cleaner::register` al tempo di vita del `Cleaner`. La struct `Cleaner` conterrà quindi un `WeakableCc` contenente la `SlotMap`, mentre i `Cleanable` conterranno un puntatore `Weak` sempre alla mappa.

La necessità di utilizzare la struttura dati `SlotMap` deriva, invece, dalla necessità di non eseguire lo stesso pulitore più di una volta. Infatti, una `SlotMap` permette di inserire dei valori generando la chiave durante l'inserimento ed assicurando chiavi univoche, anche successivamente ad una rimozione. Questo permette, in maniera ef-

ficiente (anche a livello di memoria), di poter chiamare il metodo `Cleanable::clean` più volte, ma senza eseguire il pulitore più di una volta: l'implementazione del metodo `Cleanable::clean` dovrà solo rimuovere il pulitore dalla mappa e, se era presente, eseguirlo.

Inoltre, per eseguire i pulitori registrati nella `SlotMap` quando una istanza di `Cleaner` viene distrutta è sufficiente inserire ogni closure all'interno di un wrapper, chiamato `CleanerFn` che, come mostrato nel codice 5.8, implementa un distruttore ma non `Trace`. Tale distruttore ottiene l'ownership del pulitore e lo esegue.

```
struct CleanerFn(Option<Box<dyn FnOnce() + 'static>>);

impl Drop for CleanerFn {
    fn drop(&mut self) {
        // Il metodo take rimpiazza Some(...) con None
        // e ritorna il vecchio valore
        if let Some(fun) = self.0.take() {
            fun();
        }
    }
}
```

Codice 5.8: Definizione di `CleanerFn`.
`CleanerFn` è definito come una struttura tupla unaria dove l'unico elemento è un `Option` contenente il pulitore boxato. Il pulitore è salvato come un trait object, in quanto la `SlotMap` deve poter contenere closure anche di tipo diverso. Inoltre, l'uso di un `Option` è necessario per ottenere l'ownership del pulitore durante l'esecuzione del distruttore.

Salvando i pulitori dentro tale wrapper, la loro esecuzione durante l'esecuzione del distruttore di `Cleaner` è automatica, in quanto verranno eseguiti durante la distruzione della `SlotMap`.

Infine, si può notare che l'implementazione dei cleaner è completamente safe ad eccezione delle implementazioni del trait `Trace` per i tipi `Cleaner`, `Cleanable` e `CleanerMap`. Tuttavia, tali implementazioni sono vuote e nessuno di tali tipi implementa un distruttore, perciò tali implementazioni sono equivalenti a quelle generate

dalla macro derivativa[7] (vedi 3.3.2) e quindi l'implementazione dei cleaner può essere considerata un esempio di utilizzo, da parte di codice safe, di rust-cc.

Siccome l'implementazione non utilizza unsafe (con l'eccezione mostrata in precedenza), l'utilizzo dei cleaner non può causare comportamento non definito.

Più nel dettaglio, i cleaner sono sound in quanto i valori catturati dalle closure non sono tracciati, quindi se un `Cc` venisse catturato non verrebbe mai collezionato fintanto che il pulitore non viene eseguito. Se venisse, invece, catturato un `Cc` al `CcBox` contenente il cleaner, l'allocazione non verrebbe mai collezionata a meno di eseguire manualmente il pulitore.

[7] Si sarebbe potuto, infatti, utilizzare la macro derivativa per generare tali implementazioni. Tuttavia, questo non è stato possibile a causa di come la macro genera il codice per evitare problemi di igiene dello spazio dei nomi, che causano il fallimento della compilazione di rust-cc se la macro venisse utilizzata al suo interno. Il codice generato, infatti, ridefinisce sempre il nome del crate rust-cc, utilizzando `extern crate rust_cc;`, per evitare che il nome `rust_cc` non si riferisca ad un crate. Questa ridefinizione, purtroppo, causa il fallimento della compilazione se utilizzata all'interno di rust-cc stesso.

Capitolo 6

Valutazione

In questo capitolo verrà effettuata una valutazione sia qualitativa che quantitativa di rust-cc, comparandolo con alcune delle alternative più rilevanti presenti nell'ecosistema Rust, trattando anche la modalità di testing della libreria.

6.1 Testing

La test suite di rust-cc presenta più di 80 test che ne testano completamente le funzionalità. Ogni commit è testato in CI tramite l'utilizzo delle GitHub Actions, che eseguono la test suite usando il tool `cargo-hack`[4]. Tale strumento permette, infatti, di eseguire tutti i test per ogni combinazione delle feature di rust-cc (vedi 3.1), permettendo un testing più completo rispetto alla sola esecuzione con alcune feature (manualmente) attivate.

Inoltre, siccome rust-cc fa uso estensivo di codice unsafe, la test suite viene anche eseguita utilizzando l'interprete Miri[16], un identificatore di comportamento non definito per programmi Rust.

6.2 Comparazione con le alternative

Verranno ora presentati i crate con i quali verrà comparato `rust-cc`, ovvero `gc`[8], `bacon-rajan-cc`[6], `safe-gc`[7] e `broom`[2].

Tutti i crate selezionati implementano un algoritmo di garbage collection single threaded e sono rilevanti nell'ecosistema rust: `gc` è uno dei garbage collector più utilizzati nell'ecosistema Rust (se non il più utilizzato), mentre `bacon-rajan-cc` implementa lo stesso algoritmo di `rust-cc`. `safe-gc` e `broom` offrono, invece, entrambi delle arene garbage collected. In particolare, è interessante il caso di `safe-gc`, implementato utilizzando solamente costrutti safe.

Si noti, infine, che nessuna delle alternative supporta i cleaner (vedi 3.7).

6.2.1 gc

`gc`[8] è un garbage collector della tipologia mark and sweep. Presenta una API simile a quella di rust-cc, ma con alcune importanti differenze:

1. non è possibile utilizzare il tipo `RefCell` all'interno dei tipi collezionabili, ma è necessario utilizzare il tipo `GcCell` al suo posto;
2. evitare di tracciare un campo è unsafe, in quanto è necessario tracciare sempre tutti i campi durante la fase di mark;
3. non supporta i weak pointer;
4. il trait `Trace` è più complesso da implementare manualmente di quello di rust-cc, mentre è quasi identico l'utilizzo della macro derivativa.

```
#[derive(Trace, Finalize)]
enum Node {
    Cons { next: Gc<Node>, prev: GcCell<Option<Gc<Node>>> },
    Nil,
}

let node = Gc::new(Node::Cons {
    next: Gc::new(Node::Nil),
    prev: GcCell::new(None),
});
```

```
let head = Gc::new(Node::Cons {
    next: node.clone(),
    prev: GcCell::new(None),
});
let Node::Cons { prev, .. } = &*node else { return };
*prev.borrow_mut() = Some(head.clone());
force_collect();
```

Codice 6.1: Esempio di utilizzo del crate `gc`.

6.2.2 bacon-rajan-cc

`bacon-rajan-cc`[6] è un cycle collector che utilizza lo stesso algoritmo impiegato da rust-cc. In particolare, è inteso essere una (quasi) esatta implementazione dell'algoritmo presentato da Bacon e Rajan[5], da qui il nome. Presenta una API simile a quella di rust-cc, ma con l'importante differenza di non supportare la finalizzazione e di non eseguire in automatico il collezionamento.

Inoltre, è presente una unsoundness, in quanto il trait `Trace` non è marcato come unsafe da implementare ed è quindi possibile ottenere comportamento non definito causato da codice safe.

```
enum Node {
    Cons { next: Cc<Node>, prev: RefCell<Option<Cc<Node>>> },
    Nil,
}

impl Trace for Node {
    fn trace(&self, tracer: &mut Tracer) {
        match self {
            Self::Cons { next, prev } => {
                next.trace(tracer); prev.trace(tracer);
            },
            Self::Nil => {},
        }
    }
}
```

```rust
let node = Cc::new(Node::Cons {
    next: Cc::new(Node::Nil),
    prev: RefCell::new(None),
});
let head = Cc::new(Node::Cons {
    next: node.clone(),
    prev: RefCell::new(None),
});
let Node::Cons { prev, .. } = &*node else { return };
*prev.borrow_mut() = Some(head.clone());
collect_cycles();
```

Codice 6.2: Esempio di utilizzo del crate `bacon-rajan-cc`.

6.2.3 safe-gc

`safe-gc`[7] è un crate che implementa una arena garbage collected utilizzando un algoritmo mark and sweep. Come dice il nome, è implementato completamente in rust safe utilizzando le strutture dati e gli smart pointer dalla libreria standard.

L'API offerta è ovviamente molto differente da quella di rust-cc:

1. presenta il tipo `Heap`, che implementa una arena di oggetti garbage collected che possono essere anche di tipo eterogeneo;
2. tutte le allocazioni devono essere effettuate chiamando il metodo `alloc` di una istanza di `Heap`;
3. esistono due tipi di puntatori: `Root`, che rappresenta una radice, e `Gc`, ovvero un puntatore garbage collected non radice;
4. i puntatori non possono essere direttamente dereferenziati, ma si devono usare i metodi `get` e `get_mut` di `Heap`, che accettano un puntatore (`Root` o `Gc`) e restituiscono l'oggetto puntato. Come zucchero sintattico per tali metodi, `Heap` implementa anche i trait `std::ops::Index` e `std::ops::IndexMut`;
5. utilizzare il corretto tipo di puntatore è compito del programmatore, ma un uso scorretto non porterà mai a comportamento non definito;
6. non presenta supporto né per la finalizzazione né per i weak pointers.

```
enum Node {
    Cons { next: Gc<Node>, prev: Option<Gc<Node>> },
    Nil,
}

impl Trace for Node {
    fn trace(&self, collector: &mut Collector) {
        match self {
            Self::Cons { next, prev: previous } => {
                collector.edge(*next);
                if let Some(previous) = *previous {
                    collector.edge(previous);
                }
            },
            Self::Nil => {},
        }
    }
}

let mut heap = Heap::new();
let end: Root<Node> = heap.alloc(Node::Nil);
let node: Root<Node> = heap.alloc(Node::Cons {
    next: end.unrooted(),
    prev: None,
});
let head = heap.alloc(Node::Cons {
    next: node.unrooted(),
    prev: None,
});
let Node::Cons { prev, .. } = heap.get_mut(node) else { return };
*prev = Some(head.unrooted());
heap.gc();
```

Codice 6.3: Esempio di utilizzo del crate `safe-gc`.

6.2.4 broom

`broom`[2] è un crate che implementa una arena garbage collected utilizzando un algoritmo mark and sweep in maniera simile a `safe-gc`, ma con la differenza fondamentale che gli oggetti allocati all'interno dell'arena possono essere di un solo tipo.

```
enum Node {
    Cons { next: Handle<Node>, prev: Option<Handle<Node>> },
    Nil,
}

impl Trace<Self> for Node {
    fn trace(&self, tracer: &mut Tracer<Self>) {
        match self {
            Self::Cons { next, prev } => {
                next.trace(tracer);
                if let Some(prev) = prev {
                    prev.trace(tracer);
                }
            },
            Self::Nil => {},
        }
    }
}

let mut heap: Heap<Node> = Heap::default();

let end: Handle<Node> = heap.insert_temp(Node::Nil);
let node: Handle<Node> = heap.insert_temp(Node::Cons {
    next: end, prev: None
});
```

```
let head: Rooted<Node> = heap.insert(Node::Cons {
    next: node, prev: None
});
let Some(Node::Cons { prev, .. }) = heap.get_mut(node) else {
    return;
};
*prev = Some(head.handle());
heap.clean();
```

Codice 6.4: Esempio di utilizzo del crate `broom`.

6.3 Benchmarks

Per valutare le differenze prestazionali tra rust-cc e le alternative scelte sono stati impiegati 4 benchmark: *stress test*, *binary trees*, *binary trees with parent pointers*[1] e *large linked list*, il cui sorgente è disponibile all'indirizzo `https://github.com/frengor/rust-cc-benchmarks`. Per ogni benchmark vengono riportati i risultati sia in forma tabellare che tramite un diagramma a violino. Sono anche riportati i risultati dell'esecuzione di rust-cc con la finalizzazione disattivata per un confronto più accurato con i crate che non la supportano.

I benchmark sono stati eseguiti utilizzando la libreria Criterion.rs[1] e i seguenti parametri: `-C opt-level=3 codegen-units=1 lto=true`.

Inoltre, per *binary trees*, *binary trees with parent pointers* e *large linked list* sono state anche comparate le performance rispetto ai puntatori `Rc` e `Arc` della libreria standard.

6.3.1 Stress test

Il benchmark *stress test* prova a simulare l'esecuzione di un programma creando un grafo orientato con $2^{15}+1$ vertici e $2^{15}+1$ archi casuali, mantenendo uno strong poin-

[1] I benchmark *stress test*, *binary trees* e *binary trees with parent pointers* sono stati originariamente tratti ed adattati dal crate `shredder`[14].

ter ad ogni vertice, che successivamente vengono gradualmente distrutti eseguendo collezionamenti ad intervalli regolari.

Crate	Tempo medio	Err.	Err. %	Cambiamento
rust-cc	17.53 ms	±0.20 ms	1.15%	-
rust-cc (no fin.)	17.44 ms	±0.19 ms	1.09%	−0.53%
gc	57.21 ms	±0.59 ms	1.03%	+226.25%
bacon-rajan-cc	20.86 ms	±0.24 ms	1.15%	+18.99%
safe-gc	73.16 ms	±0.35 ms	0.47%	+317.26%
broom	67.19 ms	±0.72 ms	1.08%	+283.19%

Tabella 6.1: Risultati di `stress test`.

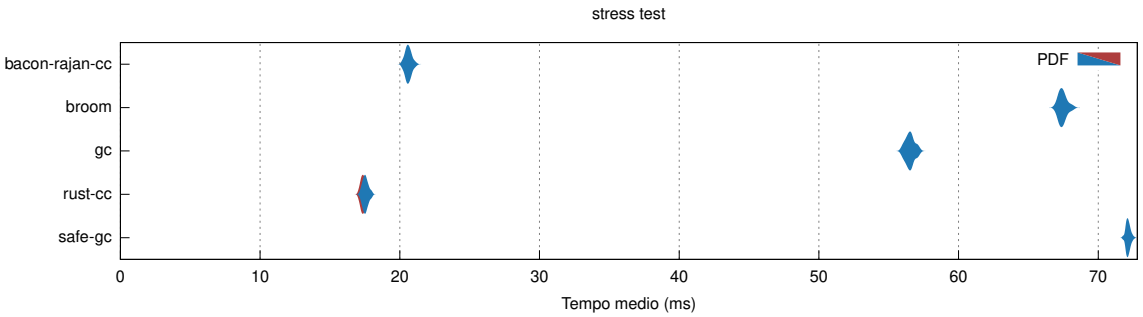


Figura 6.1: Diagramma a violino dei risultati di `stress test`.
In rosso, i risultati di rust-cc con finalizzazione disattivata.

Sia `rust-cc` che `bacon-rajan-cc` risultano ∼3 volte più veloci delle alternative, con `rust-cc` più rapido di qualche millisecondo. La differenza in prestazioni è probabilmente dovuta alla capacità dei due cycle collector di evitare di tracciare l'intero heap durante ogni collezionamento.

6.3.2 Binary trees

Il benchmark *binary trees* è pensato per misurare l'efficienza e l'impatto del reference counting in assenza di cicli. Vengono creati e distrutti alberi binari completi, senza puntatori ai nodi padri, di altezza massima 10.

Crate	Tempo medio	Err.	Err. %	Cambiamento
rust-cc	5.80 ms	±0.00 ms	0.05%	-
rust-cc (no fin.)	5.52 ms	±0.00 ms	0.03%	−4.99%
rc	4.76 ms	±0.00 ms	0.02%	−18.00%
arc	4.84 ms	±0.00 ms	0.05%	−16.70%
gc	12.90 ms	±0.03 ms	0.21%	+122.17%
bacon-rajan-cc	5.16 ms	±0.00 ms	0.08%	−11.13%
safe-gc	14.67 ms	±0.02 ms	0.12%	+152.69%
broom	28.00 ms	±0.15 ms	0.53%	+382.34%

Tabella 6.2: Risultati di `binary trees`.

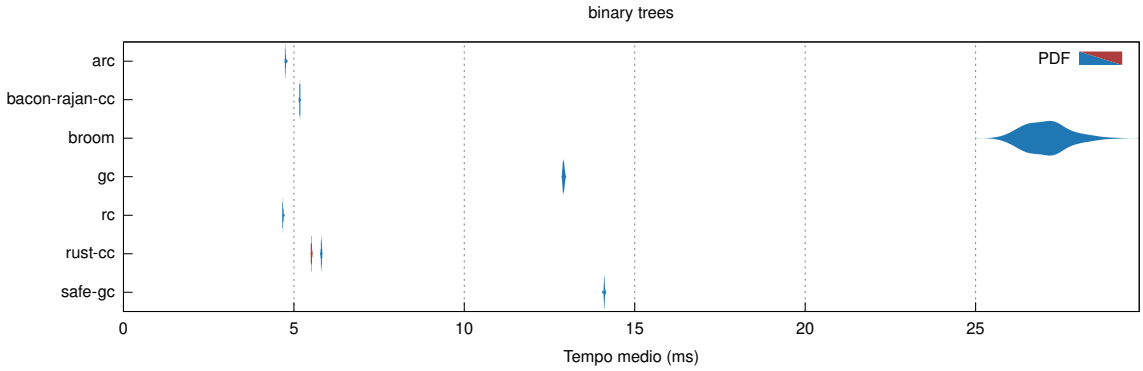


Figura 6.2: Diagramma a violino dei risultati di `binary trees`.
In rosso, i risultati di rust-cc con finalizzazione disattivata.

Gli algoritmi che presentano reference counting risultano, prevedibilmente, più veloci delle alternative e tutti molto vicini in termini di prestazioni.

6.3.3 Binary trees with parent pointers

Il benchmark *binary trees with parent pointers* è simile a *binary trees*, ma vengono aggiunti i puntatori ai nodi padri al fine di creare cicli di referenze. Ogni nodo interno degli alberi generati partecipa quindi in tre cicli, due con i figli e uno con il nodo padre. Tutti i puntatori interni agli alberi sono strong pointer ad eccezione

dei benchmark per `Rc` ed `Arc`, che utilizzano weak pointer verso i nodi padri dal momento che, altrimenti, non collezionerebbero i cicli di referenze.

Crate	Tempo medio	Err.	Err. %	Cambiamento
rust-cc	15.08 ms	±0.02 ms	0.14%	-
rust-cc (no fin.)	10.08 ms	±0.01 ms	0.12%	−33.14%
rc	4.48 ms	±0.01 ms	0.29%	−70.28%
arc	5.70 ms	±0.01 ms	0.19%	−62.23%
gc	15.13 ms	±0.02 ms	0.11%	+0.28%
bacon-rajan-cc	23.27 ms	±0.26 ms	1.11%	+54.29%
safe-gc	19.67 ms	±0.01 ms	0.07%	+30.43%
broom	32.55 ms	±1.09 ms	3.33%	+115.78%

Tabella 6.3: Risultati di `binary trees with parent pointers`.

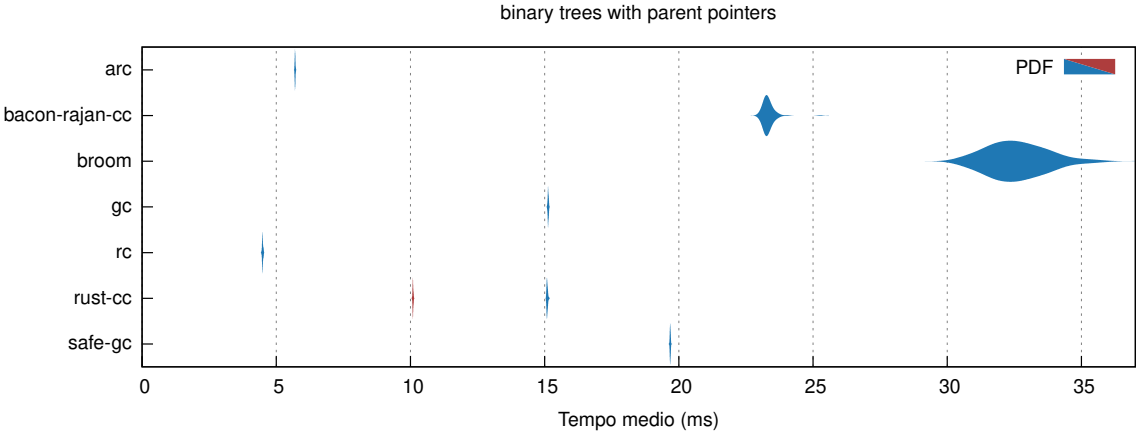


Figura 6.3: Diagramma a violino dei risultati di `binary trees with parent pointers`. In rosso, i risultati di rust-cc con finalizzazione disattivata.

`Rc` e `Arc` sono prevedibilmente i più veloci, seguiti da `rust-cc` e `gc` che impiegano entrambi ∼15 millisecondi. `bacon-rajan-cc`, nonostante impieghi lo stesso algoritmo di `rust-cc`, risulta più lento del 54%. `broom` risulta il più lento, mentre `safe-gc` si colloca tra `rust-cc` e `bacon-rajan-cc`.

Si noti, inoltre, che disabilitare la finalizzazione migliora del 33% le performance di rust-cc.

6.3.4 Large linked list

Il benchmark *large linked list* crea varie liste doppiamente concatenate contenenti 4096 elementi ciascuna e le colleziona. Come per *binary trees with parent pointers*, tutti i puntatori utilizzati sono strong pointer ad eccezione dei benchmark per `Rc` e `Arc`, che utilizzano weak pointer per i puntatori verso i nodi precedenti.

Crate	Tempo medio	Err.	Err. %	Cambiamento
rust-cc	9.21 ms	±0.01 ms	0.13%	-
rust-cc (no fin.)	6.29 ms	±0.01 ms	0.21%	−31.77%
rc	2.96 ms	±0.01 ms	0.26%	−67.83%
arc	3.89 ms	±0.00 ms	0.08%	−57.83%
gc	6.89 ms	±0.02 ms	0.33%	−25.27%
bacon-rajan-cc	14.90 ms	±0.11 ms	0.73%	+61.69%
safe-gc	13.84 ms	±0.01 ms	0.10%	+50.24%
broom	24.06 ms	±0.84 ms	3.51%	+161.15%

Tabella 6.4: Risultati di `large linked list`.

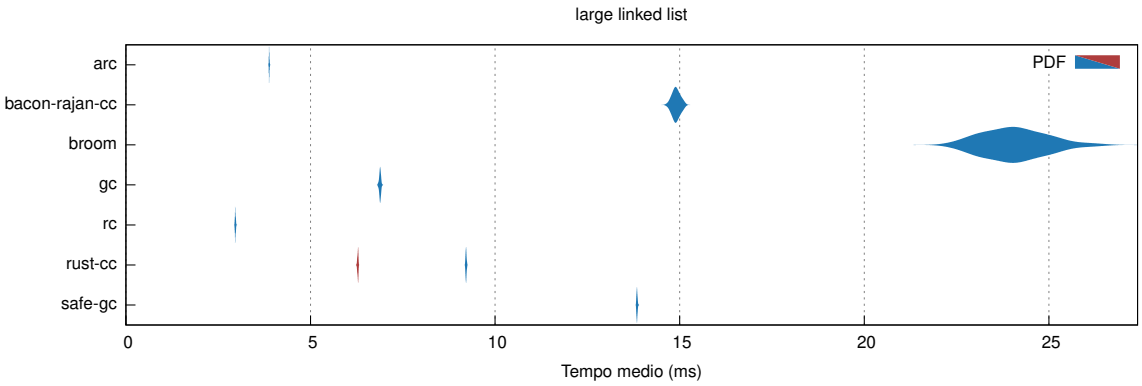


Figura 6.4: Diagramma a violino dei risultati di `large linked list`.
In rosso, i risultati di rust-cc con finalizzazione disattivata.

`Rc` e `Arc` sono ancora i più veloci, seguiti da `gc` che risulta più veloce rispetto a `rust-cc` del 25%. `bacon-rajan-cc` risulta più lento del 61% rispetto a `rust-cc`, mentre `safe-gc` si colloca sempre tra `rust-cc` e `bacon-rajan-cc`. `broom` risulta ancora il più lento.

Infine, disabilitare la finalizzazione migliora del 31% le performance di rust-cc.

Capitolo 7

Conclusioni

In questa tesi è stato presentato il crate `rust-cc`, che implementa un algoritmo efficiente di cycle collection per collezionare i cicli di referenze, e le soluzioni alle sfide incontrate durante la sua realizzazione. In termini di API, l'utilizzo di un algoritmo di cycle collection apporta dei vantaggi rispetto ad altre tipologie di garbage collector, come il mark and sweep, principalmente la possibilità di non tracciare alcuni campi in maniera safe, dal momento che non è necessario mantenere il root set per effettuare il collezionamento.

`rust-cc` è anche risultata una delle librerie più efficienti tra quelle analizzate, complice sicuramente la capacità di non dover tracciare l'intero heap ad ogni collezionamento, dovendo solamente eseguire un tracciamento (locale) a partire dagli elementi marcati come appartenenti a possibili cicli spazzatura. Inoltre, è stato possibile notare che disabilitare la finalizzazione ha sempre portato ad un incremento delle prestazioni anche significativo (fino a un 30%) ed è quindi suggeribile disabilitarla nel caso non venga utilizzata o si possa rimpiazzare il suo uso con i cleaner.

Infine, dal momento che non viene utilizzata memoria heap supplementare durante i collezionamenti, la libreria si presta bene anche all'utilizzo su target no-std come i dispositivi embedded.

7.1 Lavori futuri

Il crate ha attualmente raggiunto la sua versione `0.3.0`. Prima di una futura release stabile `1.0.0` si potrebbero sicuramente apportare alcune migliorie.

In primis, l'aggiunta di supporto ai programmi multithreaded, ad esempio tramite l'aggiunta di uno smart pointer `Acc` (Atomic Cycle Collected), il cui nome deriva da `Cc` per analogia con `Rc` e `Arc`.

In secondo luogo, `rust-cc` non dà supporto al meccanismo delle generazioni, che permetterebbe di ridurre le pause dovute alla garbage collection nel caso di grandi heap molto connessi.

Infine, per facilitarne l'utilizzo nell'implementazione di macchine virtuali, si potrebbero aggiungere alcuni metodi di utility (anche unsafe) atti a risolvere specifiche esigenze implementative di tali progetti.

Riferimenti bibliografici

[1] Aparicio, Jorge. Criterion.rs. `https://crates.io/crates/criterion`. Acceduto il 19-06-2024.

[2] Barretto, Joshua. Broom. `https://crates.io/crates/broom`. Acceduto il 20-06-2024.

[3] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, dec 1960.

[4] Taiki Endo. cargo-hack. `https://crates.io/crates/cargo-hack`. Acceduto il 29-05-2024.

[5] David F. Bacon and V.T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. *Proc. European Conf. on Object-Oriented Programming*, 2072, 2001.

[6] Fitzgerald, Nick. bacon-rajan-cc. `https://crates.io/crates/bacon-rajan-cc`. Acceduto il 20-06-2024.

[7] Fitzgerald, Nick. safe-gc. `https://crates.io/crates/safe-gc`. Acceduto il 20-06-2024.

[8] Goregaokar, Manish. gc. `https://crates.io/crates/gc`. Acceduto il 20-06-2024.

[9] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience*, 20(1):5–12, 1990.

[10] Rafael D. Lins. Cyclic Reference Counting With Lazy Mark-Scan. *Information Processing Letters*, 44(4):215–220, 1992.

[11] J. Harold McBeth. On the reference counter method. *Commun. ACM*, 6(9):575, sep 1963.

[12] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, apr 1960.

[13] Oracle and/or its affiliates. Java Platform, Standard Edition, Version 9 API Specification - Cleaner. `https://docs.oracle.com/javase%2F9%2Fdocs%2Fapi%2F%2F/java/lang/ref/Cleaner.html`. Acceduto il 20-06-2024.

[14] Peach, Gregor. shredder. `https://crates.io/crates/shredder`. Acceduto il 19-06-2024.

[15] Orson Peters. slotmap. `https://crates.io/crates/slotmap`. Acceduto il 21-06-2024.

[16] Miri team. Miri. `https://github.com/rust-lang/miri`. Acceduto il 29-05-2024.

[17] The Rust Foundation. Rust. `https://www.rust-lang.org/`. Acceduto il 21-06-2024.

Appendici

Appendice A

Approfondimento sull'incompatibilità tra crate causata da supertrait richiesti solamente con specifiche feature abilitate

Nel paragrafo 3.4 è stato spiegato come sia sempre necessario implementare il trait `Finalize` insieme a `Trace`, in quanto il primo è supertrait del secondo, anche quando feature relativa alla finalizzazione (`finalization`) è disabilitata.

Questa scelta è dovuta ad un caso di incompatibilità tra crate che si presenta quando un trait è supertrait di un altro solamente quando una specifica feature è abilitata.

Infatti, prendiamo in esame il caso mostrato nella figura A.1, dove è mostrato l'albero delle dipendenze di un ipotetico crate `my-crate`.

`my-crate` dipende da due crate: `crate-A` e `crate-B`, che dipendono entrambi da `rust-cc`. La feature `finalization` è abilitata soltanto dal crate `crate-A`.

Siccome le feature di Cargo sono addittive, quando entrambi i crate `crate-A` e `crate-B` vengono compilati assieme, la feature `finalization` verrà abilitata anche per il secondo crate! Questo causerebbe quindi il fallimento della compilazione di

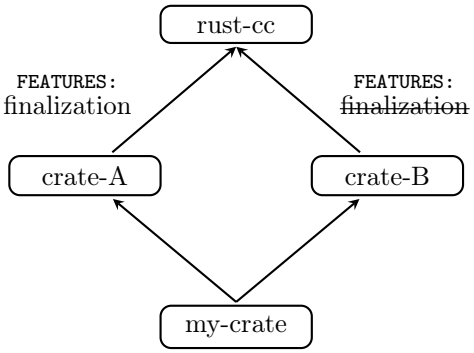


Figura A.1: Albero delle dipendenze del crate `my-crate`.

`crate-B`, in quanto il trait `Finalize` non risulterebbe implementato, anche se tale crate non riporta problemi quando compilato in maniera a sé stante.

Appendice B

Casi limite della finalizzazione

Solitamente un collezionamento necessita di eseguire soltanto due volte l'identificazione dei cicli prima di poter procedere con il liberamento della memoria, in quanto i finalizzatori vengono eseguiti successivamente alla prima, mentre il liberamento della memoria avviene dopo la seconda. Tuttavia, come mostrato nel paragrafo 5.3, questo non accade sempre e ci sono delle misure per prevenire che il collezionamento (o la fase di finalizzazione) possa non terminare.

Tali misure prevengono la non terminazione, anche se è comunque possibile che programmi accuratamente scritti possano fare spendere molto tempo al collezionamento. Ne è un esempio il codice B.1: dopo ogni finalizzazione di una istanza di `Evil` e finché il vettore `VEC` non è vuoto, esiste sempre una istanza di `Evil` che diventa finalizzabile.

```
#[derive(Trace)]
struct Evil {
    // Evil deve far parte di un ciclo di referenze
    cyclic: RefCell<Option<Cc<Evil>>>,
    ...
}
```

```
impl Evil {
    fn new() -> Cc<Evil> {
        // Crea e restituisce una istanza di
        // Evil dove cyclic forma un cappio
        ...
    }
}

thread_local! {
    static VEC: RefCell<Vec<Cc<Evil>>> = RefCell::new(vec![]);
}

impl Finalize for Evil {
    fn finalize(&self) {
        // Un elemento viene rimosso ad ogni finalizzazione
        let _ = VEC.with_borrow_mut(Vec::pop);
    }
}

// Aggiungi 1000 elementi a VEC
VEC.set((0..1000).map(|_| Evil::new()).collect());

// Distruggi il primo Evil e colleziona
let _ = Evil::new();
// Il collezionamento eseguirebbe 1000 volte
collect_cycles();
```

Codice B.1: Esempio di caso limite della finalizzazione.

Il codice in esame aggira, quindi, la limitazione posta per evitare casi come il codice 5.4, ovvero che ogni valore allocato durante la fase di finalizzazione è considerato come già finalizzato.

Crediamo che sia difficile che un programma reale possa accidentalmente presentare un tale comportamento. Tuttavia, è comunque importante evitare che un tale caso interrompa l'esecuzione del programma utilizzatore di rust-cc troppo a lungo fermandolo nella fase di collezionamento. Per questo motivo, è stato posto il limite di 10 esecuzioni successive massime dell'identificazione dei cicli. I `CcBox` non collezionati saranno inseriti nuovamente all'interno di `POSSIBLE_CYCLES` per poter essere collezionati in futuro.

Questo è un trade-off tra collezionare sempre tutti gli elementi dei cicli spazzatura (che presenta il problema appena mostrato) ed eseguire l'algoritmo sempre al massimo due volte. Infatti, è comunque possibile che un programma necessiti alcune esecuzioni supplementari. Eseguire l'algoritmo dieci volte significa supportare meglio tali programmi dando, comunque, una durata massima al collezionamento.

Ringraziamenti

Ringrazio innanzitutto i docenti del corso “Linguaggi di programmazione” che, con le loro affascinanti lezioni, mi hanno ispirato a scrivere il primo prototipo di garbage collector che ha poi dato origine a rust-cc.

Ringrazio particolarmente il Prof. Saverio Giallorenzo, che mi ha seguito durante la stesura della tesi e per tutti i suoi preziosissimi consigli.

Ringrazio i miei genitori Fabio e Giusi per essere sempre stati al mio fianco ed avermi sostenuto nei momenti difficili.

Non può mancare un ringraziamento speciale a mio fratello Damiano, che mi ha sempre voluto bene e che con le sue parole mi ha infuso fiducia e sicurezza.

Ringrazio i miei nonni e tutti i parenti per aver sempre creduto in me nel percorso per arrivare a questo primo traguardo.

Ringrazio con affetto tutti i miei amici, persone fantastiche con le quali ho condiviso momenti bellissimi e tantissime risate. In particolare, ringrazio Davide, con il quale condivido anche moltissimi progetti.

Infine, ringrazio Matthew House (`https://github.com/LegionMammal978`) per la sua contribuzione nella quale ha sostituito l'`UnsafeCell` all'interno di `CounterMarker` con una `Cell` safe.