Modern Algorithms for Garbage Collection

Outlining modern algorithms for garbage collection on the examples of Go and Java

6424019, 9525469

December 4, 2023

Contents

This paper is aimed to introduce the reader to the concept of garbage collection. Outlining modern algorithms for garbage collection, their scope and strategies while comparing garbage collection with other memory management techniques. The goal is to explain introduced ideas using examples written in *Go* and *Java* as well as highlighting their garbage collection implementations and the ideas and theorems behind them while contrasting trade-offs either implementations had to make.

1 Introduction

Garbage collection refers to the process of automatically managing heap allocated memory on behalf of the running process by identifying parts of memory that are no longer needed. This is often performed by the runtime of a programming language while the program is executing. [1, Introduction] [2, Introduction]

Most programming languages allocate values with static lifetimes[1] in main memory along with the executable code. Values that are alive for a certain scope are allocated using the call stack[2] without requiring dynamic allocation. These Variables can't escape the scope they were defined in and must be dynamically allocated if accessing them outside of their scope is desired.

This requires the programmer to allocate and deallocate these variables to prevent memory leaks[3] provided the programming language does not perform garbage collection.

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    char *name;
    double age;
} Person;

Person *new_person(char *name, double age) {
  Person *p = malloc(sizeof(Person));
  p->age = age, p->name = name;
  return p;
}
// [...]
```

Listing 1.1: C heap allocation

Listing Listing 1.1 showcases a possible use case for dynamic memory allocation. The `Person` structure is filled with values defined in the parameters of the `new_person` function. This structure, if stack allocated, would not live longer than the scope of the `new_person` function, thus rendering this function useless. To create and use a `Person` structure outside of its scope, the structure has to be dynamically allocated via the `malloc` function defined in the `#include <stdlib.h>` header.

[1] variable available for the whole runtime of the program [3, Abstract]
[2] stores information about running subroutines / functions [4, 2.2 Call Stacks]
[3] allocated no longer needed memory not deallocated [5, 1.2.1 A Practical Object Ownership Model]

```c
// [...]
int main(void) {
    for(int i = 0; i < 1e5; i++) {
        Person *p = new_person("max musterman", 89);
    }
    return EXIT_SUCCESS;
}
```

Listing 1.2: C heap allocation with memory leakage

See listing Listing 1.2 for an example of a memory leak. Here the program creates $1 * 10^5$ `Person` structures using the `new_person` function allocating each one on the heap but not releasing their memory after the iteration ends and therefore rendering the reference to them inaccessible, which generally defines a memory leakage this programming error can lead to abnormal system behaviour and excessive RAM consumption in long lived applications [6, Description]. The definitive solution for memory leaks is determining leaking variables and freeing them, see listing Listing 1.3.

```c
// [...]
int main(void) {
    for(int i = 0; i < 1e5; i++) {
        Person *p = new_person("max musterman", 89);
        free(p);
    }
    return EXIT_SUCCESS;
}
```

Listing 1.3: C heap allocation without memory leakage

Another potential issue with manual memory management is accessing already released memory classified as *use-after-free* errors [7]. Consider the modified example in listing Listing 1.4 showcasing value access of a `Person` structure after its memory has already been released.

```c
// [...]
int main(void) {
    for(int i = 0; i < 1e5; i++) {
        Person *p = new_person("max musterman", 89);
        free(p);
        printf("Person{name: '%s', age: %f}\n", p->name, p->age);
    }
    return EXIT_SUCCESS;
}
```

Listing 1.4: C heap allocation with freed memory access

The example in listing Listing 1.4 results in undefined behaviour [7, Description] and could cause crashes if memory the program can not legally access is accessed, could cause memory corruption if the memory region pointed to contains data after the previous data has been released or could be exploited to inject data into the application [7, Consequences].

```
// [...]
void *free_person(Person *p) {
  free(p);
  return NULL;
}

int main(void) {
    for(int i = 0; i < 1e5; i++) {
        Person *p = new_person("max musterman", 89);
        p = free_person(p);
        if(p == NULL) continue;
        printf("Person{name: '%s', age: %f}\n", p->name, p->age);
    }
    return EXIT_SUCCESS;
}
```

Listing 1.5: C heap allocation without freed memory access

A common resolution for this issue is setting a pointer to `NULL` via `p = NULL` and checking if the pointer is `NULL` before accessing it (see listing Listing 1.5) [7, Related Controls].

Garbage collection manages dynamically allocated memory for the programmer, therefore issues such as memory leakages and accessing released memory can be prevented by not exposing capabilities for manual memory management. A language such as golang contains a garbage collector [2, Introduction] enabling automatically releasing no longer used memory blocks, as shown in listing Listing 1.6. The garbage collector in listing Listing 1.6 automatically deallocates the result of `new_person` upon it leaving the scope of the loop iteration it was called in.

```go
package main

import "fmt"

type Person struct {
    Name string
    Age  float64
}

func NewPerson(name string, age float64) *Person {
    return &Person{name, age}
}

func main() {
    for i := 0; i < 1e5; i++ {
        p := NewPerson("max musterman", 89)
        fmt.Printf("Person{name: %q, age: %f}\n", p.Name, p.Age)
    }
}
```

Listing 1.6: Go allocation example

2 Garbage Collection

As introduced before (see chapter 1) the process of garbage collection is required by many programming languages via their specification, as is the case with *Java* [8, Chapter 1. Introduction] and *Go* [2, Introduction]. The *Go* programming language specification however does not include specifics around the implementation of its garbage collection [1, Introduction]. The *Go* standard tool chain provides a runtime library included in all executables created by the *Go* compiler. This library contains the garbage collector [1, Introduction].

Garbage collection as a whole is an umbrella term for different concepts, algorithms and ideas. This chapter includes the differentiation between these and thereby introduces terms necessary for understanding the following chapters.

2.1 Scope

The scope of garbage collection refers to the variables, resources and memory areas it manages. Garbage collection is generally responsible for managing already allocated memory, either by the programmer or the libraries / subroutines the programmer uses [9, Abstract]. The aforementioned can be cumulated to heap allocated memory or dynamically allocated memory. This represents the purview of a garbage collector [10, 1 Introduction]. The listing Listing 2.1 showcases variables that will be garbage collected upon the scope of the `GarbageCollected.main()` function ends.

```
class Scope {
    static class Test {
    }

    public static void main(String[] args) {
        var test1 = new Scope.Test();
        var test2 = new Scope.Test();
    }
}
```

Listing 2.1: Java variables managed by the garbage collector

The areas not managed by the garbage collector and thus not in the scope of this paper are open resources requiring being closed by the consumer (such as sockets or `java.util.Scanner` [11, close]) and stack allocated variables as well as statically allocated variables. The listing Listing 2.2 displays a variety of variables not garbage

collected due to all of them being stack allocated primitive types [8, 4.2. Primitive Types and Values].

```
class Scope1 {
    static int integer = 5;

    public static void main(String[] args) {
        byte newline = 0x1A;
        double pi = 3.1415;
        char a = 'a';
    }
}
```

Listing 2.2: Java variables not managed by the garbage collector

2.2 Strategies

Garbage collection can be implemented using a variety of strategies, each differing in their code complexity, RAM/CPU usage and execution speed [12, 4.3 Benchmarks] [13, Motivation and Historical Perspective].

2.2.1 Tracing

Most commonly the term garbage collection is used to refer to tracing garbage collection. This strategy of automatically managing memory is a common way of implementing garbage collection. Tracing is defined as determining which objects should be deallocated. This is done by tracing which of the currently allocated objects are accessible via linked references. Accessible objects are marked as alive. Memory regions not accessible via this list are not marked and therefore considered to be unused memory and are deallocated. [13, Garbage Collection Background]

Programming languages such as *Java* [14, 2.2 Full GC algorithm], *Go* [1, Tracing Garbage Collection] and *Ocaml* [15, Garbage Collection, Reference Counting, and Explicit Allocation] use this strategy for deallocating unused memory regions.

2.2.1.1 Categorizing memory

Objects[1] are categorised as reachable or alive if they are referenced by at least one variable in the currently running program, see Listing 2.3 for a visualisation. This includes references from other reachable objects. As introduced before, the definition of tracing garbage collection includes determining whether or not objects are reachable. In the paragraph above, this reachability is defined. This definition does not include the objects the tracing garbage collector refers to as *root*-objects [16, Preliminaries:

[1] Dynamically allocated memory region containing one or more values [1, Tracing Garbage Collection]

Heap Depth and Tracing]. root-objects are defined as generally accessible, such as local variables, parameters and global variables.[2] Root-objects are used as a starting point for tracing allocated objects [16, Preliminaries: Heap Depth and Tracing].

In Listing 2.3, both values initially assigned to `x` and `y` in the `Main.main` function are considered inaccessible due to the reassignment of `x` and `y` in the following lines. The value of the variable `z` in the `Main.f` function is considered inaccessible once the scope of the function ends, when the variable `z` is dropped from the call stack - rendering its value inaccessible.

```
public class MemoryCategories {
    public static void main(String[] args) {
        var x = new Object();
        x = new Object();
        var y = new Object();
        y = new Object();
        MemoryCategories.f();
    }

    private static void f() {
        var z = new Object();
    }
}
```

Listing 2.3: Java example for accessible and inaccessible memory

2.2.1.2 Implementations

As introduced before the main idea behind tracing garbage collection is to trace the memory set[3]. Garbage collection is often performed in cycles. Cycles are triggered when certain conditions are met, such as the program running out of memory and therefore not being able to satisfy an allocation request or the cycles are ran on a predefined interval. The process of tracing memory and deallocating memory require separation, they are therefore often split into different garbage collection cycles. The following concepts and implementation details can be and are generally intertwined in modern garbage collectors [1, The GC cycle] [15].

2.2.1.2.1 Mark and Sweep Garbage collectors using the *mark and sweep*-concept traverse the object graph[4] starting from the root-objects, therefore satisfying the definition of a tracing garbage collector, as presented in Section 2.2.1.1. The main detail of the mark and sweep technique is marking each encountered object of the object graph as alive. This stage of the process is referred to as *marking*. The stage defined

[2] As introduced in Section 2.1: variables on the call stack or static variables
[3] Virtual memory the program makes use of
[4] Objects and pointers to objects

as *sweeping* entails walking over the memory on the heap and deallocating all non marked objects [1, Tracing Garbage Collection].

2.2.1.2.2 Generational Generational garbage collection is based on the empirical observation that recently allocated objects are most likely to be inaccessible quickly[5]. Objects are differentiated into generations, this is often implemented by using separate memory regions for different generations. Upon filling a generations memory region its objects are being traced by using the older generation as roots, this usually results in most objects of the generation being deallocated. The remaining objects are moved into the older generations memory region [10, 2 Age-based Garbage Collection]. This technique results in fast incremental garbage collection, considering one memory region at a time is required to be collected. [10, 3 Benchmarks]

2.2.1.2.3 Stop the world Stop the world garbage collector refer to the process of halting the execution of the program for running a garbage collection cycle. Therefore guaranteeing that no new objects are allocated or becoming unreachable while performing the garbage collection cycle. The main advantage of this implementation approach is that it introduces less code complexity while being faster than the previously introduced incremental garbage collection [17, 5. The Garbage Collection Algorithms]. This technique is inherently unsuited for applications requiring real-time performance, such as games or web servers in which unexpected latency has drastic results.

2.2.2 Reference Counting

Reference counting garbage collection is defined as each object keeping track of the amount of references made to it. This reference counter is incremented for each created reference and decremented for each destroyed reference. Once the counter reaches 0 the object is no longer considered reachable and therefore deallocated [18, 2.2 Precise Reference Counting] [19, 6. Reference Counting Automatic Storage Reclamation Algorithms].

In contrast to the previously introduced tracing garbage collection this approach promises that objects are immediately deallocated once their last reference is destroyed. Due to the reference count being attached to their respective objects this strategy is CPU cache friendly [18, 1. Introduction].

Reference counting garbage collection has several disadvantages to the aforementioned tracing garbage collection. These can be mitigated via sophisticated algorithms. The following chapters highlight a selection of problems commonly occurring when implementing reference counting garbage collection [18, 2. Overview].

[5] Generally known as *infant mortality* or *generational hypothesis*

2.2.2.1 Memory usage

Reference counting requires attaching a reference counter onto allocated objects, thus increasing the overall memory footprint proportionally to the amount of allocated objects and a reference counter for each object.

$$
\begin{aligned}
n &:= \text{Amount of Objects} \\
m &:= \text{Object size} \\
r &:= \text{Reference counter size}
\end{aligned}
$$

Memory footprint without reference counting:

$$nm$$

Memory footprint with reference counting:

$$nm + nr$$

2.2.2.2 Cycles

Two or more objects creating references to each other is described as a reference cycle. This results in none of the objects being categorised as garbage as their collective references never let their reference count decrement to 0.

A way to prevent reference cycles is by extending reference counting garbage collection to specifically detecting cycles, as is the case in *CPython* [20, 1.10 Reference Counts].

2.2.2.3 Increment and Decrement Workload

Each reference creation and reference falling out of scope requires modification of the reference count of one or more objects [19, 6.1 Immediate Reference Counting]. There are methods for decreasing this workload, such as ignoring stack references to objects until they are about to be deallocated, triggering a stack scan for making sure the object is no longer referenced [19, 6.2 Deferred Reference Couting] or merging reference counter modifications [21, Abstract].

2.2.2.4 Thread safety

Reference counting garbage collection requires atomic operations in multithreaded environments to keep a consistent count of references. This requires expensive overhead and is often mitigated with a reference counter per thread. This solution introduces significant memory overhead and is not commonly used [21, 1.2 Reference-Counting on a Multiprocessor].

2.2.3 Escape Analysis

The term *escape analysis* describes a compile-time technique for determining where to store an object, either on the heap or the stack. At a high-level the analysis determines whether an allocated object is reachable outside of its current scope. If so the object is said to *escape* to the heap. Otherwise the object is allocated on the stack and as previously introduced deallocated/dropped once the scope ends. [22, `l.18`]. Due to the omitted cost of managing the short lived allocated objects not used outside of their scope, the workload of the garbage collector is reduced significantly [1, Escape analysis].

```
type T struct { x int64 }

func A() *T {
    return &T{x:12}
}

func B() {
    t := &T{x:25}; t.x++
}
```

Listing 2.4: Go example for escape analysis

In Listing 2.4 the allocated structure of type `*T` in function `A` escapes to the heap due to the fact that it is returned from `A`. The structure assigned to `t` of type `*T` in `B` is dropped upon the `t.x++` instruction is executed and the scope of `B` ends. The Go compiler allocates the value of `t` on the stack - a direct result of escape analysis [22].

3 Comparison with other Memory Management Techniques

In this section alternatives to garbage collection for memory management are presented and compared to garbage collection.

3.1 Manual Memory Management

Manual memory management is the most basic memory management technique. It is used in languages like C and C++. In this technique the programmer is responsible for allocating and freeing memory. This is done by calling the `malloc` and `free` functions in C and the `new` and `delete` operators in C++. The programmer has to keep track of the allocated memory and free it when it is no longer needed. This is done by storing the pointer returned by the allocation function in a variable and passing it to the free function when it is no longer needed. This is illustrated in Listing 3.1.

```
int main() {
    // Allocate memory for a single integer
    int* a = malloc(sizeof(int));
    *a = 42;

    // Allocate memory for an array of 10 integers
    int* b = malloc(sizeof(int) * 10);
    for (int i = 0; i < 10; i++) {
        b[i] = i;
    }

    // Free the allocated memory
    free(a);
    free(b);
}
```

Listing 3.1: Example of manual memory management in C

This technique is very error prone and can lead to memory leaks and use-after-free errors resulting in undefined behaviour and security vulnerabilities as explained in chapter 1. However it is usually the fastest memory management technique because it does not have any overhead compared to garbage collection.

3.2 Lifetimes and Borrow Checking

The desire for the performance of manual memory management and the safety of garbage collection has led to the development of a new memory management technique called *lifetimes and borrow checking*. The main idea behind this technique is that the corresponding `free` calls for heap memory can be automatically inserted at compile time by the compiler, if the compiler can prove that the memory is no longer needed. When a variable is no longer needed, it is said to have reached the end of its *lifetime* hence the name of the technique.

Because this is run at compile-time the performance is similar to manual memory management. The safety is comparable to garbage collection because the compiler can prove that there are no use-after-free errors or memory leaks when compiling, assuming the compiler is correct. While this technique has the best-of-both-worlds properties of manual memory management and garbage collection for safety and performance, it lacks in ease-of-use because the programmer has to follow a set of rules. Satifying these rules can be difficult and can take sometimes take a time, especially for beginners [23]

This memory management technique in the presented form was first introduced in the *Rust* programming language [24, 1. Introduction] replacing the garbage collector it initially had [25]. Because Rust was the first language to implement this concept, the examples in this section will be written in Rust.

3.2.1 Ownership

The first step to understand this technique is to understand the concept of *ownership*.

In Rust, every value is always owned by exactly a variable inside a scope. When the variable goes out of scope, the value is dropped. The ownership of a value can be transferred to another variable by *moving* it. This can be either in the form of an assignment or as a function return value. When a value is moved, the previous owner can no longer access the value. When a value is dropped by going out of scope, any memory it owns is freed, including heap memory [26, pp. 59–61].

A major contrast in Rust compared to other programming languages like C is that variable assignments like `let a = b` are moving the value instead of copying it [24, 2.2 Ownership]. The same goes for the parameter values for function calls. Because of this variables can not be used after being used in a variable assignment or function call.

A example showcasing the ownership concept similar [1] to the C example presented in

[1] The presented Rust example differs from the C example because the `create_person` function does not return a reference but a value. However the struct consists of a `String` which is a heap allocated dynamic length string that gets allocated by the `.into()` call converting the static `&str` into `String`, so the example still requires the used heap memory to be freed. References/Borrows to temporary values are not allowed in Rust so the only way to force a heap allocation of the whole struct would be to use a `std::boxed::Box<T>`. This was not used in this example for legibility reasons.

the introduction can be found in Listing 3.2 [2].

```
struct Person {
    name: String,
    age: f64,
}

fn new_person(name: String, age: f64) -> Person {
    Person { name, age }
}

fn print_person(person: Person) {
    println!("{} is {} years old.", person.name, person.age);
}

fn main() {
    let person = new_person("Rainer Zufall".into(), 42.0);
    let person1 = person; // value of person is moved to person1
    // print_person(person); // error: use of moved value: `person`
    print_person(person1);

    { // Example of sub-scope
        let person2 = new_person("Jona Zufall".into(), 13.0);
        print_person(person2);
    } // person2 is dropped here
} // person1 and person are dropped here
```

Listing 3.2: Person struct example in Rust demonstrating ownership

3.2.2 Borrowing

The second step to understand this technique is to understand the concept of *borrowing*. Allowing only one owner to access a variable at a time would be too restrictive for many uses cases.

As a example calling the `print_person` function twice on the same person would not be possible, because the ownership of the person would be moved to the function after the first call and cannot be accessed anymore. This is illustrated in Listing 3.3.

```
let person = new_person("Rainer Zufall".into(), 42.0);
print_person(person);
print_person(person); // error: use of moved value: `person`
```

Listing 3.3: Failed attempt to print a person twice in Rust due to lost ownership

[2] A more idomatic Rust implementation would define the new and print functions as methods of the Person struct. This was not done here to keep the example simple for readers not familiar with Rust.

The solution to this problem is the concept of borrowing. It essentially is the pointer concept from C and other languages but with the ownership model of Rust in mind, which imposes some restrictions on it. Borrowing allows a value owner to give another function or struct access to a value without giving ownership to the function [24, 2.3 Borrowing]. A owner can hand out many read-only borrows to a value at the same time, but only one mutable borrow at a time. This is done to avoid data races but is not strictly needed for the memory management aspect of the technique [26, p. 90].

A modified version of the presented person example to make use of borrows can be found in Listing 3.4.

```
fn print_person(person: &Person) {
    println!("{} is {} years old.", person.name, person.age);
}

fn main() {
    let person = new_person("Rainer Zufall".into(), 42.0);
    print_person(&person); // borrow using &
    print_person(&person); // borrow a second time
} // person is dropped here
```

Listing 3.4: Person struct example in Rust demonstrating borrowing

A borrow can only be used as long as the owner is still alive. A borrow can not outlive the owner variable. This is enforced at compile time using the borrow checker. Through this free-after-use errors can be detected at compile time. This also means that the timepoint in program execution when a heap allocated value is no longer needed is always when the owner variable lifetime ends because there cannot be any borrows to the value after that point [26, p. 188] [24, 2.4 Reference Lifetimes] [27].

3.2.3 Multi-owner values using reference counters

Some usecases require a value to be owned by multiple owners at the same time. These usecases include shared memory and cyclic data structures.

Allowing multiple owners for a single value can be done by using reference counters as a escape hatch. A reference counter is a data structure that keeps track of the number of owners of a value and drops the value when the number of owners reaches zero. This is done by incrementing the counter when a new owner is created using `.clone()` and decrementing it when a owner is dropped. Reference counter implementations are available in the Rust standard library as `std::rc::Rc<T>` [26, pp. 320–323] and `std::sync::Arc<T>` [26, p. 361] for single threaded and atomic multi-threaded use respectively. An example of this can be found in Listing 3.5.

The cloned reference counter instance can be moved to other owners and outlive the original instance. This works great for non-cyclic data structures, but not so well by itself for cyclic data structures (see Section 2.2.2.2) because the reference counter will never reach zero and the memory will never be freed. To solve this problem, you can

```rust
use std::rc::Rc;

fn main() {
    // Create an Rc that contains a person
    let person = Rc::new(new_person("Rainer Zufall".into(), 42.0));

    // Clone the Rc to create additional references
    // These can be moved to other owners and outlive the original Rc instance
    let clone1 = Rc::clone(&person);
    let clone2 = Rc::clone(&person);

    println!("Reference count of person: {}", Rc::strong_count(&person));

    // Access the data through the cloned references
    println!("clone1 data: {:?}", clone1);
    println!("clone2 data: {:?}", clone2);

    // When the references go out of scope, the reference count decreases
    drop(clone1);
    println!("Count after dropping clone1: {}", Rc::strong_count(&person));

    drop(clone2);
    println!("Count after dropping clone2: {}", Rc::strong_count(&person));

    // At this point, the reference count drops to zero, and the memory is
    // deallocated because the last reference is dropped.
}

// output:
// Reference count of person: 3
// clone1 data: Person { name: "Rainer Zufall", age: 42.0 }
// clone2 data: Person { name: "Rainer Zufall", age: 42.0 }
// Count after dropping clone1: 2
// Count after dropping clone2: 1
```

Listing 3.5: Reference counter example in Rust

use weak references [26, pp. 334–335] or break the cycle manually when you are done with the data structure. When the developer does not deal with this problem a memory leak will occur.

4 Garbage collected Programming Languages

In this chapter, the used garbage collection implementations of two programming languages are presented. These are implementations of the theoretical concepts presented in chapter 2

4.1 Go

4.2 Java

Java by default uses a generational garbage collector as introduced in paragraph 2.2.1.2.2. This garbage collector is called Garbage First (G1) and was introduced in Java 9 [28]. Before that, Java used various types of mark and sweep collectors [29].

Beyond those there are many more garbage collectors available for Java that can be used by specifying them as a command line argument to the JVM. These are not relevant for this writing, as they are not used by default. Nonetheless these can be very useful when wanting to use a garbage collector tuned to a specific use case.

Contrary to the theoretical concept of a generational garbage collector, the memory areas for each generation in G1GC are not continuous in memory. Instead G1GC uses a heap divided into regions usually 1 MB - 32 MB in size. Each region is assigned to one of the generations. These generations are called Eden, Survivor and Old. [30]

Bibliography

[1] "A guide to the go garbage collector." (2022), [Online]. Available: https://tip.golang.org/doc/gc-guide (visited on 10/09/2023).

[2] "The go programming language specification." (2023), [Online]. Available: https://tip.golang.org/ref/spec (visited on 10/11/2023).

[3] "Beyond static and dynamic scope." (2009), [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1837513.1640137 (visited on 10/11/2023).

[4] "Call stack coverage for gui test-suite reduction." (2006), [Online]. Available: https://www.cs.umd.edu/~atif/papers/McMasterMemonISSRE2006.pdf (visited on 10/11/2023).

[5] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive c and c++ memory leak detector," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 168–181.

[6] "Memory leak." (2020), [Online]. Available: https://owasp.org/www-community/vulnerabilities/Memory_leak (visited on 10/12/2023).

[7] "Using freed memory." (2020), [Online]. Available: https://owasp.org/www-community/vulnerabilities/Using_freed_memory (visited on 10/12/2023).

[8] "Java language specification." (2023), [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/jls-1.html (visited on 10/19/2023).

[9] L. Cen, R. Marcus, H. Mao, J. Gottschlich, *et al.*, "Learned garbage collection," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2020, pp. 38–44.

[10] D. Stefanović, K. S. McKinley, and J. E. B. Moss, "Age-based garbage collection," in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, 1999, pp. 370–381. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/320384.320425.

[11] "Class scanner." (2006), [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html#close-- (visited on 10/19/2023).

[12] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 25–36, 2004. [Online]. Available: https://users.cecs.anu.edu.au/~steveb/pubs/papers/mmtk-sigmetrics-2004.pdf.

[13] M. Maas, K. Asanović, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 138–151. [Online]. Available: https://people.eecs.berkeley.edu/~krste/papers/GC-MicroTopPicks2019.pdf.

[14] P. Pufek, H. Grgić, and B. Mihaljević, "Analysis of garbage collection algorithms and memory management in java," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 1677–1682. DOI: 10.23919/MIPRO.2019.8756844.

[15] "Garbage collection." (2023), [Online]. Available: https://ocaml.org/docs/garbage-collection (visited on 10/09/2023).

[16] K. Barabash and E. Petrank, "Tracing garbage collection on highly parallel platforms," in *Proceedings of the 2010 International Symposium on Memory management*, 2010, pp. 1–10. [Online]. Available: https://web.archive.org/web/20150131070633id_/http://www.cs.technion.ac.il:80/~erez/Papers/parallel-trace-ismm.pdf.

[17] G. L. Steele, "Multiprocessing compactifying garbage collection," *Commun. ACM*, vol. 18, no. 9, pp. 495–508, Sep. 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361005. [Online]. Available: https://doi.org/10.1145/361002.361005.

[18] A. Reinking, N. Xie, L. de Moura, and D. Leijen, "Perceus: Garbage free reference counting with reuse," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 96–111. [Online]. Available: https://xnning.github.io/papers/perceus.pdf.

[19] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," *ACM Sigplan notices*, vol. 19, no. 5, pp. 157–167, 1984. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/390011.808261.

[20] "Extending and embedding the python interpreter." (2008), [Online]. Available: https://docs.python.org/release/2.5.2/ext/refcounts.html (visited on 11/06/2023).

[21] Y. Levanoni and E. Petrank, "An on-the-fly reference-counting garbage collector for java," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 1, pp. 1–69, 2006. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1111596.1111597.

[22] The Go Authors. "Escape analysis." (2018), [Online]. Available: https://cs.opensource.google/go/go/+/master:src/cmd/compile/internal/escape/escape.go (visited on 11/06/2023).

[23] W. Crichton, *The usability of ownership*, 2021. arXiv: 2011.06171 [cs.PL].

[24] D. J. Pearce, "A lightweight formalism for reference lifetimes and borrowing in rust," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 1, pp. 1–73, Mar. 2021. DOI: 10.1145/3443420. [Online]. Available: https://doi.org/10.1145/3443420.

[25] P. Walton, *Removing garbage collection from the rust language*, 2013. [Online]. Available: https://web.archive.org/web/20230911102334/https://pcwalton.github.io/_posts/2013-06-02-removing-garbage-collection-from-the-rust-language.html (visited on 11/13/2023).

[26] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, Aug. 2019, ISBN: 9781718500440.

[27] D. Drysdale. “Effective rust: Understanding the borrow checker.” (2023), [Online]. Available: https://www.lurklurk.org/effective-rust/borrows.html (visited on 11/09/2023).

[28] H. Grgic, B. Mihaljević, and A. Radovan, “Comparison of garbage collectors in java programming language,” in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018, pp. 1539–1544. DOI: 10.23919/MIPRO.2018.8400277.

[29] Oracle and/or its affiliates. “Java virtual machine garbage collection - available collectors.” (2017), [Online]. Available: https://docs.oracle.com/javase/9/gctuning/available-collectors.htm.

[30] “Getting started with the g1 garbage collector.” (2021), [Online]. Available: https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html (visited on 11/17/2023).