Modern Algorithms for Garbage Collection

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

Daniel Huber Matteo Gropp

October 13, 2023

Contents

1 Introduction **3**

2 Overview over Garbage Collection **7**

3 Comparison with other Memory Management Techniques **8**
3.1 Manual Memory Management 8
3.2 Lifetimes and Borrow Checking 8

4 Tradeoffs between Garbage Collection Algorithms **9**

5 Overview over current Garbage Collection Algorithms **10**
5.1 Go 10
5.2 Python 10
5.3 OCaml 10
5.4 Java 10

Abstract

Summary in english and in german here

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. [1, Introduction] This is often performed by the runtime of a programming language while the program is executing. [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: C heap allocation

Listing 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 struc-

[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]

ture has to be dynamically allocated via the `malloc` function defined in the `#include <stdlib.h>` header.

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

Listing 2: C heap allocation with memory leakage

See listing 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 3.

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

Listing 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 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 4: C heap allocation with freed memory access

The example in listing 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].

```c
// [...]
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 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 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 6.

```
package main

import "fmt"

type Person struct {
    Name string
    Age  float64
}

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

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

Listing 6: Go allocation example

The garbage collector in listing 6 automatically deallocates the result of `new_person` upon it leaving the scope of the loop iteration it was called in.

2 Overview over Garbage Collection

3 Comparison with other Memory Management Techniques

3.1 Manual Memory Management

3.2 Lifetimes and Borrow Checking

4 Tradeoffs between Garbage Collection Algorithms

5 Overview over current Garbage Collection Algorithms

5.1 Go

5.2 Python

5.3 OCaml

5.4 Java

References

[1] *A Guide to the Go Garbage Collector*. URL: https://tip.golang.org/doc/gc-guide (visited on 10/09/2023).

[2] *The Go Programming Language Specification*. URL: https://tip.golang.org/ref/spec (visited on 10/11/2023).

[3] *Beyond static and dynamic scope*. URL: https://dl.acm.org/doi/abs/10.1145/1837513.1640137 (visited on 10/11/2023).

[4] *Call Stack Coverage for GUI Test-Suite Reduction*. URL: https://www.cs.umd.edu/~atif/papers/McMasterMemonISSRE2006.pdf (visited on 10/11/2023).

[5] David L Heine and Monica 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*. URL: https://owasp.org/www-community/vulnerabilities/Memory_leak (visited on 10/12/2023).

[7] *Using freed memory*. URL: https://owasp.org/www-community/vulnerabilities/Using_freed_memory (visited on 10/12/2023).