

Masterarbeit

## **Titel der Masterarbeit**

Phil Steinhorst

|                                    |                             |
|------------------------------------|-----------------------------|
| <i>Erstgutachter und Betreuung</i> | Prof. Dr. Jan Vahrenhold    |
| <i>Zweitgutachter</i>              | Prof. Dr. Markus Müller-Olm |

Münster, 14. November 2018

**Titel der Masterarbeit**

Masterarbeit zur Erlangung des akademischen Grades *Master of Education*  
in den Fächern Mathematik und Informatik

Erstgutachter und Betreuung: Prof. Dr. Jan Vahrenhold

Zweitgutachter: Prof. Dr. Markus Müller-Olm

Münster, 14. November 2018

**Phil Steinhorst**

Dürerstraße 1, 48147 Münster

p.st@wwu.de

Matrikelnummer: 382 837

**Westfälische Wilhelms-Universität Münster**

Fachbereich 10 – Mathematik und Informatik

Institut für Informatik

Einsteinstraße 62, 48149 Münster

# Zusammenfassung

Die vorliegende Arbeit soll eine Aufarbeitung verschiedener Ansätze für Garbage-Collection-Algorithmen liefern. Nach einer kurzen Darstellung der zugrunde liegenden Problematik und deren praktische Relevanz sowie den Vor- und Nachteilen einer automatischen Speicherverwaltung gegenüber einer manuellen Speicherverwaltung werden gängige Ansätze vergleichend vorgestellt sowie Einsatz und Eignung in der Praxis beurteilt. Als Gütekriterien dienen hier beispielsweise Laufzeitbetrachtungen, Speicherbedarf und entstehende Verzögerungen im Programmablauf, die für ausgewählte Ansätze besonders detailliert untersucht werden.

Weiter wird eine Anwendung entworfen, mit der die Arbeitsweise der diskutierten Garbage-Collection-Ansätze visualisiert werden kann. Dazu gehört eine angemessene Visualisierung eines beschränkten Speicherbereichs, etwa durch eine optische Unterscheidbarkeit belegter Blöcke, sowie der einzelnen Arbeitsphasen, die eine Garbage Collection ausführt. Dabei sollen auch unterschiedliche Szenarien auswählbar sein, etwa verschiedene Speicherfüllstände und eine variable Anzahl bzw. Größe von Objekten, die im Speicher hinterlegt sind.

Am Ende nochmal schauen, ob das wirklich so ist :D

## Abstract

Englisch einfügen.



# Inhaltsverzeichnis

|           |   |           |
|-----------|---|-----------|
| <b>1</b>  | <b>Einleitung</b>   | <b>1</b>  |
| 1.1       | Problemstellung und Terminologie . . . . .                          | 1         |
| <b>I</b>  | <b>Algorithmen und Ansätze</b>                                      | <b>3</b>  |
| <b>2</b>  | <b>Mark and Sweep</b>   | <b>5</b>  |
| 2.1       | Naives Mark and Sweep . . . . .                                     | 5         |
| <b>II</b> | <b>Entwurf und Realisierung eines Garbage-Collection-Simulators</b> | <b>9</b>  |
| <b>3</b>  | <b>Modellierung</b>   | <b>11</b> |
|           | <b>Anhang</b>   | <b>13</b> |
| <b>A</b>  | <b>Test</b>   | <b>15</b> |
|           | <b>Literatur</b>  | <b>17</b> |
|           | <b>Abbildungsverzeichnis</b>  | <b>19</b> |
|           | <b>Tabellenverzeichnis</b>  | <b>21</b> |
|           | <b>Algorithmenverzeichnis</b>                                       | <b>23</b> |
|           | <b>Eigenständigkeitserklärung</b>                                   | <b>25</b> |



# Einleitung

Einleitung: Was ist Garbage Collection, wozu ist das relevant?

## 1.1 Problemstellung und Terminologie

Was ist das Ziel einer GC? Wie kann man das möglichst formal ausdrücken? Grundbegriffe und Modellierung des Speichers?





# Teil I

---

Algorithmen und Ansätze



# Mark and Sweep

Wir beginnen mit einer Vorstellung des ersten Garbage-Collection-Algorithmus, der auf John MCCARTHY zurückgeht [McC60, S. 191–193]. Im Rahmen eines im Jahr 1960 veröffentlichten Artikels über die Berechnung rekursiver Funktionen auf dem *IBM 704* mithilfe des *LISP Programming Systems* erläutert McCarthy die Speicherung von Daten in einer Listenstruktur. Diese besteht aus Paaren, deren erster Eintrag *car* die zu speichernde Information enthält, während im zweiten Eintrag *cdr* die Registeradresse des nachfolgenden Paares zu finden ist. Register, die aktuell nicht zur Speicherung von Daten genutzt werden, befinden sich in einer *free storage list*. Bei der Anforderung von Speicher für ein zu speicherndes Datum werden Register aus dieser Liste entfernt. Durch die Manipulation der Registeradressen können Paare verwaisen, was zu Speicherlecks führt. Zur Auflösung dieser Problematik bietet LISP als erste Programmiersprache ihrer Zeit eine automatische Speicherverwaltung, die von McCarthy wie folgt grob umschrieben wird: Im Falle von Speicherknappheit wird – ausgehend von einer Menge von Basisregistern – ermittelt, welche Register über eine Folge von *cdr*-Einträgen erreichbar sind. Nicht erreichbare Register enthalten überschreibbare Inhalte, sodass diese zurück in die *free storage list* eingefügt werden können und wieder als freie Speicherplätze zu Verfügung stehen. Diese zweischrittige Vorgehensweise – das Erkennen nicht mehr benötigter Speicherbereiche und die anschließende Freigabe eben jener – bildet die Grundlage des *Mark-and-Sweep*-Algorithmus.

## 2.1 Naives Mark and Sweep

Der naive Mark-and-Sweep-Algorithmus arbeitet in zwei Schritten: Zunächst wird bestimmt, welche Objekte im Speicher unerreichbar sind, weil sie von keinem anderen erreichbaren Objekt referenziert werden. Diese Objekte können gefahrlos freigegeben werden, da auf ihre Informationen nicht mehr zugegriffen werden kann. Der zweite Schritt besteht aus einer Traversierung des gesamten Heaps. Dabei werden alle existierenden Objekte besucht und diejenigen freigegeben, die im ersten Schritt als unerreichbar identifiziert werden konnten. Die entsprechenden Speicherbereiche stehen anschließend wieder für neue Objekte zu Verfügung.

---

**Algorithmus 2.1** Naives Mark and Sweep – Markierung (vgl. [JL96, Kap. 2.2])

---

```
1: collectGarbage():  
2:   markStart()  
3:   sweepHeap()  
4:  
5: markStart():  
6:   todo  $\leftarrow \emptyset$                                 ▷ Noch abzuarbeitende Objekte  
7:   for each obj  $\in$  ROOTS                               ▷ Beginne mit Basisobjekten  
8:     if (isNotMarked(obj))  
9:       setMarked(obj)                                ▷ Objekt als erreichbar markieren  
10:      add(todo, obj)  
11:      mark()                                          ▷ Abarbeitung starten  
12:  
13: mark():  
14:   while todo  $\neq \emptyset$   
15:     obj  $\leftarrow$  remove(todo)                        ▷ Hole nächstes Objekt  
16:     for each adr  $\in$  POINTERS(obj)                 ▷ Hole nächste Referenz auf Objekt  
17:       if (adr  $\neq$  null && isNotMarked(*adr))  
18:         setMarked(*adr)  
19:         add(todo, *adr)
```

---

Die Markierungsphase (engl. *mark*) funktioniert wie folgt: Zunächst wird mittels der Methode *markStart* eine Menge *todo* erzeugt, die diejenigen Objekte enthält, die bereits als erreichbar erkannt wurden, aber selbst noch nicht verarbeitet wurden (Zeile 6 in Algorithmus 2.1). In diese werden alle bislang unmarkierten Basisobjekte der Menge *ROOTS* eingefügt und markiert, da sie in jedem Fall erreichbar sind (Zeile 7 bis 10). Ist ein Basisobjekt bereits markiert worden, so wurde es schon entdeckt – etwa, weil es durch ein zuvor abgearbeitetes Objekt referenziert wird. Daraus folgt, dass es ebenfalls bereits abgearbeitet wurde oder sich noch in der Menge *todo* befindet. In beiden Fällen muss es folglich nicht erneut zu *todo* hinzugefügt werden.

Bereits nach dem Hinzufügen des ersten Basisobjekts wird die Methode *mark* aufgerufen, welche die *todo*-Menge abarbeitet. Für jedes Objekt in *todo* werden diejenigen Felder betrachtet, die eine Referenz auf ein Objekt enthalten (Zeile 15 und 16). Wenn dieses Objekt noch nicht markiert wurde, wird es in diesem Augenblick zum ersten Mal entdeckt. Da es somit erreichbar ist, kann es markiert und zu *todo* hinzugefügt werden, um zu einem späteren Zeitpunkt abgearbeitet zu werden (Zeile 17 und 18). Verweist die Referenz hingegen auf ein Objekt, das bereits markiert wurde, wurde dieses schon zuvor entdeckt. Auch hier ist ein erneutes Hinzufügen zu *todo* überflüssig. Sobald *todo* leer ist, erfolgt die Rückkehr zur Methode *markStart*, sodass ggfs. das nächste Basisobjekt abgearbeitet wird.

Kleines Beispiel hier, großes in den Anhang?

Es ist wesentlich, dass Objekte bereits markiert werden, wenn sie der Menge `todo` hinzugefügt werden, und nicht etwa, nachdem sie abgearbeitet wurden (Zeile 9 und 10 bzw. 18 und 19). Andernfalls besteht bei zyklischen Referenzen die Gefahr einer Endlosschleife, da unmarkierte Objekte mehrfach hinzugefügt würden. Präziser können wir festhalten, dass `todo` zu jedem Zeitpunkt ausschließlich bereits markierte Objekte enthält. Da keine Objekte hinzugefügt werden, die bereits markiert wurden (Zeile 8 und 16), wird kein Objekt doppelt verarbeitet. Da zudem mit jeder Iteration der **while**-Schleife mindestens ein Objekt aus `todo` entfernt wird (Zeile 15), die Anzahl aller Objekte endlich ist und wir voraussetzen, dass während der Ausführung des Garbage Collectors keine neuen Objekte entstehen, wird sowohl die **while**-Schleife, als auch die **for each**-Schleife nach endlich vielen Schritten terminieren.

Anstatt die Abarbeitung der `todo`-Menge zu beginnen, sobald das erste Basisobjekt erfasst wurde, können statt dessen auch zunächst alle Basisobjekte zu `todo` hinzugefügt und die Methode `mark` im Anschluss aufgerufen werden. Je nachdem, wie `todo` in der Praxis realisiert wird – zum Beispiel in Form eines Stacks – kann damit die Traversierung der Objekte beeinflusst werden. Dies kann einen erheblichen Einfluss auf die Performanz der Markierungsphase haben, wenn Caching-Effekte eine Rolle spielen.

---

**Algorithmus 2.2** Naives Mark and Sweep – Bereinigung (vgl. [JL96, Kap. 2.2])

---

```
1: sweep():  
2:   pos ← nextObject(HEAP_START)  
3:   while pos ≠ null  
4:     if isMarked(*pos)  
5:       unsetMarked(*pos)  
6:     else free(pos)  
7:     pos ← nextObject(pos)
```

---

evtl. später  
genauer  
drauf  
eingehen

Die Bereinigungsphase (engl. *sweep*) beginnt unmittelbar nach der Markierungsphase durch Aufruf der Methode *sweep*. Die Variable `pos` wird mit der Speicheradresse initialisiert, an der sich das erste Objekt im Heap befindet. Wir gehen davon aus, dass eine Methode *nextObject* zu Verfügung steht, die anhand einer übergebenen Speicheradresse die Adresse des nachfolgenden Objektes oder `null` zurückgibt, wenn dieses nicht existiert. Dadurch wird der Heap linear traversiert; nicht markierte Objekte werden freigegeben, während die Markierung erreichbarer Objekte zurückgesetzt wird.

Markierungsmöglichkeiten (Tricolor, Bitmapping); Lazy Sweeping



# Teil II

---

Entwurf und Realisierung eines  
Garbage-Collection-Simulators





# Modellierung

3

Designentscheidungen



# Anhang

---



Test

A

blablu



# Literatur

- [JL96] Richard Jones und Rafael Lins. *Gargabe Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester: John Wiley & Sons, 1996 (zitiert auf den Seiten 6, 7).
- [McC60] John McCarthy. „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I“. In: *Communications of the ACM* 3.4 (1960), S. 184–195 (zitiert auf Seite 5).

Diese Masterarbeit wurde mit  $\text{\LaTeX}$  unter Verwendung der Vorlage *Clean Thesis* von Ricardo Langner gesetzt. Für mehr Informationen siehe <http://cleanthesis.der-ric.de/>.





# Abbildungsverzeichnis



## Tabellenverzeichnis



# Algorithmenverzeichnis

|     |   |   |
|-----|---|---|
| 2.1 | Naives Mark and Sweep – Markierung . . . . .  | 6 |
| 2.2 | Naives Mark and Sweep – Bereinigung . . . . . | 7 |



# Eigenständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Masterarbeit *Titel der Masterarbeit* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

---

(Ort, Datum)

---

(Unterschrift)