Westfälische Wilhelms-Universität Münster
Fachbereich 10 – Mathematik und Informatik
Institut für Informatik

Masterarbeit

Titel der Masterarbeit

Phil Steinhorst

Erstgutachter und Betreuung Prof. Dr. Jan Vahrenhold

Zweitgutachter Prof. Dr. Markus Müller-Olm

Münster, 16. 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, 16. 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

1 Einleitung

Die Möglichkeiten einer dynamischen Speicherverwaltung haben sich in den meisten modernen Programmiersprachen etabliert. Die Vorteile, einen Teil des dynamischen Speichers – oft auch als *Heap* bezeichnet – zur Laufzeit eines Programms anfordern zu können, sind unbestreitbar: Speicherbereiche, die zum Heap gehören, dienen als Ablagemöglichkeit für Unterprogramme jenseits ihrer eigenen *Stacks*, sodass ihre Inhalte nach Terminierung erhalten und für weitere Unterprogramme zugänglich bleiben. Die Größe des angeforderten Speichers muss dabei nicht zur Compilezeit bekannt sein, was die Realisierung dynamischer Datenstrukturen ermöglicht und die Überschreitung hartkodierter Speicherbereiche vermeidet.

Für die konkrete Verwendung einer dynamischen Speicherverwaltung sind grundsätzlich zwei diametrale Ansätze denkbar: Zum einen kann die Verantwortung für den korrekten Umgang mit dynamisch angefordertem Speicher gänzlich der Entwicklerin übertragen werden. Dies ist in der Regel mit zusätzlichem Aufwand verbunden: Speicheradressen müssen manuell verwaltet werden, Anweisungen zur Anforderung und Freigabe von Speicher müssen in den eigentlichen Code integriert werden und entsprechende Ausnahmefälle bei Fehlschlägen müssen ordnungsgemäß abgefangen werden. Neben einer komplexer werdenden Codestruktur führt dies zu weiteren Fehlerquellen: Die Freigabe noch benötigten Speichers führt zu so genannten *hängenden Zeigern* (engl. *dangling pointer*) – Referenzen, die *ins Leere zeigen* und in der Folge bestenfalls zu Programmabstürzen, schlimmstenfalls aber zu unerwartetem Verhalten und Datenverlust führen können. Nicht freigegebener, aber nicht mehr benötigter Speicher kann wiederum zu *Speicherlecks* (engl. *memory leaks*) und – bei hinreichend langer Laufzeit des Programms – zu einer Ausschöpfung des Speichers führen. *Double frees*, bei denen Speicherbereiche doppelt freigegeben werden, sind eine weitere Ursache für unerwünschtes Programmverhalten. Während die Anforderung von Speicher in der Regel unproblematisch ist, ist die Frage, wann und an welcher Stelle angeforderter Speicher wieder freigegeben werden kann, deutlich komplizierter, und fehlerhafte Verwendungen werden gegebenenfalls erst bei langfristiger Ausführung des Programms bemerkt.

Zum anderen existiert zur Vermeidung eben jener Schwierigkeiten der Ansatz, dem Compiler und der Laufzeitumgebung die adäquate Freigabe nicht mehr benötigten Speichers zu überlassen. Zuständig hierfür ist dann ein Mechanismus, der gemeinhin

als *Garbage Collection* (engl. für *Abfallentsorgung*) bezeichnet wird. Eine Garbage Collection führt automatisch zu bestimmten Zeitpunkten – etwa regelmäßig oder wenn akuter Speichermangel besteht – eine Bereinigung des Speichers durch und gibt nicht mehr benötigte Speicherbereiche frei, ohne dass der Entwickler entsprechende Routinen in sein Programm integrieren muss. Nichtsdestoweniger wird dieser Komfortgewinn nicht ohne Nachteile erworben: Wie jede Programmanweisung besitzt auch eine Garbage Collection einen gewissen Bedarf an Rechenzeit und Ressourcen, der sich negativ auf die Performance der eigentlichen Anwendung auswirken kann. Vor allem in Anwendungen, die einen hohen Durchsatz erreichen wollen oder in denen Deadlines um jeden Preis eingehalten werden müssen, spielt die Auswahl eines geeigneten Garbage-Collection-Algorithmus eine nicht unerhebliche Rolle.

In dieser Arbeit werden wir gängige Ansätze zur Garbage Collection vorstellen und miteinander vergleichen. Dabei soll auch ein Augenmerk auf Performance und Ressourcenbedarf gelegt sowie die Eignung in verschiedenen Anwendungsfällen beurteilt werden. Im zweiten Teil der Arbeit wird der Entwurf und die Implementation einer Anwendung beschrieben, die die diskutierten Garbage-Collection-Algorithmen grafisch visualisiert und in einem vereinfachten Speichermodell simuliert. Anhand dieser Anwendung soll die Arbeitsweise der Algorithmen veranschaulicht werden.

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

2 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 ← ∅                                  ▷ Noch abzuarbeitende Objekte
 7:     for each obj ∈ 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 ≠ ∅
15:         obj ← remove(toDo)                    ▷ Hole nächstes Objekt
16:         for each adr ∈ POINTERS(obj)          ▷ Hole nächste Referenz auf Objekt
17:             if (adr ≠ 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.

evtl. später genauer drauf eingehen

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)
```

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.

Wir halten zunächst fest, dass der Mark-and-Sweep-Algorithmus in seiner Gänze terminiert und korrekt ist, sofern während der Garbage Collection das laufende Programm angehalten wird:

Satz 2.1:
Der Mark-and-Sweep-Algorithmus terminiert und ist korrekt, wenn der Mutator während der Arbeit des Kollektors angehalten wird.

Beweis: Wie oben erläutert, terminiert die Markierungsphase in jedem Fall, da bei angehaltenem Mutator keine neuen Objekte erstellt werden. Gleiches gilt für die Bereinigungsphase, in der alle Objekte des Heaps in endlicher Zeit besucht werden.

Korrektheit

□

Die Bedingung, dass der Mutator während des Markierens pausiert wird, ist tatsächlich notwendig, um vermeiden, dass fälschlicherweise keine erreichbaren Objekte entfernt werden, wie folgendes Beispiel zeigt (vgl. [Dij+78, S. 969]): Betrachten wir etwa die Situation, dass zwei Basisobjekte A und B alternierend auf ein Objekt C verweisen, das ausschließlich über A oder B erreichbar ist. Während der Kollektor aktiv ist, führe der Mutator folgenden Code aus:

kleine Grafik einfügen

```
1: B.ref ← &C
2: A.ref ← null
3: A.ref ← &C
4: B.ref ← null
```

Es könnte passieren, dass der Kollektor gerade Objekt A abarbeitet, unmittelbar nachdem Zeile 2 ausgeführt wurde. Es wird dann keine Referenz auf Objekt C vorgefunden. Wenn der Kollektor nun Objekt B betrachtet, nachdem bereits Zeile 4 abgearbeitet wurde, wird Objekt C weiterhin nicht entdeckt. Insgesamt wird Objekt C somit nicht markiert, obwohl es erreichbar ist. In der Folge würde C irrtümlich freigegeben werden, sodass im schlimmsten Fall ein hängender Zeiger entsteht oder sogar Datenverlust verursacht wird – der Algorithmus arbeitet also nicht korrekt.

Algorithmen, die zur Vermeidung von *race conditions* zwischen Kollektor und Mutator die Arbeit des letzteren unterbrechen, werden auch als *Stop-the-World-Algorithmen* bezeichnet.

Markierungsmöglichkeiten (Tricolor, Bitmapping); Lazy Sweeping

Teil II

Entwurf und Realisierung eines
Garbage-Collection-Simulators

3 Modellierung

Designentscheidungen

Anhang

Test

A

blablubb

Literatur

[Dij+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten und E. F. M. Steffens. „On-the-fly Garbage Collection: An Exercise in Cooperation“. In: *Communications of the ACM* 21.11 (1978), S. 966–975 (zitiert auf Seite 8).

[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: *Commununications of the ACM* 3.4 (1960), S. 184–195 (zitiert auf Seite 5).

Diese Masterarbeit wurde mit LaTeX 2$_\varepsilon$ unter Verwendung der Vorlage *Clean Thesis* von Ricardo Langner gesetzt. Für mehr Informationen siehe `http://cleanthesis.der-ric.de/`.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

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)