

Todo list

Am Ende nochmal schauen, ob das wirklich so ist :D	v
Englisch einfügen.	v
JV: In welchen Sprachen nicht?	1
s. JV	1
JV: evtl. auf WCET in EmbSys hinweisen?	2
evtl genauer eingehen	3
Grafik zu Heap und Objekten einfügen	3
definieren	5
Kleines Beispiel hier, großes in den Anhang?	12
evtl. später genauer drauf eingehen	13
Markierungsmöglichkeiten (Tricolor, Bitmapping); Lazy Sweeping	18
Designentscheidungen	21

Masterarbeit

Garbage-Collection-Algorithmen und ihre Simulation

Phil Steinhorst

Erstgutachter und Betreuung

Prof. Dr. Jan Vahrenhold

Zweitgutachter

Prof. Dr. Markus Müller-Olm

Münster, 28. November 2018

Garbage-Collection-Algorithmen und ihre Simulation

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, 28. 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	1
1.1	Terminologie	2
1.2	Problemstellung	4
I	Algorithmen und Ansätze	9
2	Mark and Sweep	11
2.1	Naives Mark and Sweep	11
2.2	Markierungsmöglichkeiten	15
2.3	Drei-Farben-Abstraktion	15
II	Entwurf und Realisierung eines Garbage-Collection-Simulators	19
3	Modellierung	21
	Anhang	23
A	Test	25
	Literatur	27
	Abbildungsverzeichnis	29
	Tabellenverzeichnis	31
	Algorithmenverzeichnis	33
	Eigenständigkeitserklärung	35

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 des Heaps dienen für Unterprogramme als Ablagemöglichkeit jenseits ihrer eigenen *Stacks*, sodass abgelegte Inhalte nach Terminierung erhalten und für weitere Unterprogramme zugänglich bleiben. Die Größe des angeforderten Speichers muss dabei nicht zur Übersetzungszeit bekannt sein, was die Realisierung dynamischer Datenstrukturen ermöglicht und die Überschreitung hartkodierter Speicherbereiche vermeidet.

JV: In welchen Sprachen nicht?

s. JV

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 (vgl. [Wil92, S. 1f]): 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** (dt. *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 die Entwicklerin entsprechende Routinen in ihr 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 signifikante Rolle.

JV: evtl.
auf
WCET in
EmbSys
hinwei-
sen?

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 Terminologie

Bevor wir genauer darauf eingehen, was unter einer Garbage Collection konkret verstanden wird, sollen zunächst die nötige Terminologie sowie ein Speichermodell eingeführt werden, das im Fortgang dieser Arbeit benutzt wird. Dieses Speichermodell ist bewusst so abstrakt gehalten, dass es möglichst allgemeine Betrachtungen lösgelöst von gängigen Programmiersprachen, Laufzeitumgebungen und Betriebssystemen ermöglicht, auch wenn an einigen Stellen exemplarisch Bezüge zu diesen hergestellt werden. Die eingeführten Begrifflichkeiten orientieren sich stark an der Terminologie aus [JL96, Kap. 1].

Objekt

Unter einem **Objekt** verstehen wir stets eine konkrete Instanz eines definierten Datentyps, beispielsweise eines struct in C oder einer Java-Klasse. Ein Objekt besitzt eine festgelegte Anzahl von **Feldern**, die jeweils einen Wert eines festgelegten Datentyps – etwa ein Integer oder eine Referenz auf ein anderes Objekt im hier definierten Sinne – enthalten. Der in dieser Arbeit verwendete Objektbegriff ist wesentlich allgemeiner gehalten als in der Objektorientierung üblich: Auch einzelne

Werte eines Basisdatentyps oder Arrays werden als Objekt aufgefasst, selbst wenn diese nicht Bestandteil eines im Programm definierten Datentyps sind.

Wir setzen ferner voraus, dass Objekte und ihre Felder *typisiert* sind. Das bedeutet, dass stets nachvollziehbar ist, aus welchen Feldern ein Objekt besteht von welchem Datentypen diese sind. Insbesondere ist unterscheidbar, ob ein Feld eines Objekts eine Referenz enthält oder nicht. Weiter nehmen wir an, dass jedes Objekt einen so genannten *Header* besitzt. Dies ist ein separates Feld, das Metainformationen aufnimmt, die für den Compiler und die Laufzeitumgebung, nicht aber aus Sicht des Entwicklers, zugänglich sind. Diverse vorgestellte Algorithmen werden diesen Bereich nutzen, um für die Speicherverwaltung relevante Informationen zu hinterlegen.

evtl genauer eingehen

Den Zugriff auf das *i*-te Feld eines Objekts *a* notieren wir – analog zur Syntax der Programmiersprache C – mit *a[i]*. Ebenso bezeichnen wir mit *&a* die Adresse eines Objekts und mit **p* die Dereferenzierung eines Zeigers *p*. Mit *POINTERS(a)* bezeichnen wir zudem die Menge aller Felder eines Objekts *a*, die eine Referenz enthalten können.

Heap

Als **Heap** bezeichnen wir denjenigen Speicherbereich, in dem zur Laufzeit eines Programms Objekte in beliebiger Reihenfolge erzeugt und freigegeben werden können. Der Heap besteht aus Blöcken einer festen Größe, auf die über eine Speicheradresse zugegriffen werden kann; ein *Block* ist dabei die kleinste zuweisbare Speichermenge und kann die Zustände *belegt* (bzw. *zugewiesen*) oder *frei* annehmen. Sofern nichts anderes vereinbart ist, gehen wir davon aus, dass der Heap ein zusammenhängender linearer Speicherbereich ist.¹

Grafik zu Heap und Objekten einfügen

Allokator, Mutator und Kollektor

Aufgabe des **Allokators**, der zur Laufzeitumgebung eines Programms gehört, ist zum einen die Zuweisung von Heapspeicher bei dynamischer Instanziierung eines neuen

¹Tatsächlich ist dies eine starke Vereinfachung. In der Praxis ist der Bereich des physikalischen Speichers, der von einer Anwendung verwendet wird, häufig fragmentiert und inhomogen. Die Speicherverwaltung eines Betriebssystems bildet diesen Bereich auf einen *virtuellen Speicher* ab, der der Anwendung zu Verfügung gestellt wird und aus ihrer Sicht linear zusammenhängend ist. Für einen Überblick hierzu siehe etwa [TB14, Kap. 3.3].

Objektes und zum anderen die Freigabe von Objekten. Der Allokator führt somit Buch über die belegten und freien Blöcke des Heaps. Die genaue Realisierung dieser Mechanismen werden in dieser Arbeit weitestgehend außen vor gelassen, jedoch setzen wir in gewissen Situationen das Vorhandensein bestimmter Funktionalitäten voraus. Beispielsweise verlangen wir, dass eine Prozedur *new* zu Verfügung steht, die bei der Erzeugung eines neuen Objekts Speicher reserviert und die entsprechende Speicheradresse zurückgibt. Die Funktionsweise von *new* kann dabei vom verwendeten Garbage-Collection-Algorithmus abhängen (siehe Algorithmus 1.1).

Algorithmus 1.1 Methode *new* zur Erzeugung eines neuen Objekts. Die Garbage Collection wird hier bei Bedarf ausgelöst, wenn nicht genügend freier Speicher verfügbar ist.

```

1: new():  

2:   adr ← allocate()           ▷ Versuche Zuweisung von Speicher  

3:   if adr = null              ▷ Nicht genügend freier Speicher  

4:     collectGarbage()         ▷ Aufruf der Garbage Collection  

5:     adr ← allocate()         ▷ Neuer Versuch  

6:     if adr = null  

7:       error("Nicht genügend Speicher")  

8:   return adr

```

Nach Dijkstra et al. besteht ein Programm zudem aus zwei funktional unterscheidbaren Bestandteilen [Dij+78, S. 967]: Der **Mutator** ist derjenige Thread (bzw. eine Menge von Threads), die den eigentlichen Programmcode ausführen. Für uns sind dabei vor allem Programmanweisungen von Bedeutung, die in Feldern von Objekten vorhandene Referenzen manipulieren und somit ursächlich für die Entstehung von nicht mehr benötigten Objekten sind. Im Gegensatz dazu ist es die Aufgabe des **Kollektors**, die nicht mehr benötigten Objekte zu identifizieren und ihre Freigabe zu veranlassen. Der Kollektor ist demnach derjenige Thread (bzw. eine Menge von Threads), die einen Garbage-Collection-Algorithmus ausführen.

1.2 Problemstellung

Nachdem die nötigen Grundbegriffe eingeführt wurden, können wir nun definieren, was wir unter einer Garbage Collection verstehen. Anschließend folgt eine Spezifikation von Eigenschaften, die wir von einem Garbage-Collection-Algorithmus fordern.

Definition 1.1 (Lebendigkeit):

Ein Objekt heißt zu einem bestimmten Zeitpunkt im Programmablauf **lebendig**, wenn der Mutator im weiteren Programmablauf lesend oder schreibend auf dieses zugreift. Andernfalls bezeichnen wir das Objekt als *nicht mehr benötigt*.

Definition 1.2 (Garbage Collection):

Eine Garbage Collection ist ein Algorithmus zur automatischen Wiederverwendung bereits genutzten Heapspeichers durch Identifikation und Freigabe von Objekten, die im weiteren Programmverlauf nicht mehr benötigt werden.

Sobald der Mutator auf eine Objektinstanz im weiteren Programmverlauf nicht mehr zugreift – weder lesend, noch schreibend – ist ein Überschreiben des Objekts unproblematisch. Demzufolge darf eine Garbage Collection die Freigabe des entsprechenden Speicherbereichs veranlassen, sobald eine Stelle im Programmcode erreicht wurde, ab der der Bezeichner eines Objekts (bzw. eine Referenz auf dieses Objekt) nicht mehr verwendet wird – auch, wenn theoretisch noch darauf zugegriffen werden könnte. Allerdings ist die Frage, ob dies der Fall ist oder nicht, nicht beantwortbar:

Satz 1.1 (Unentscheidbarkeit von Lebendigkeit):

Es existiert kein Algorithmus, der die Lebendigkeit von Objekten entscheidet.

Beweis: Dies ist ein Korollar aus der Unentscheidbarkeit des Halteproblems: Angenommen, es gäbe einen Algorithmus, der für ein beliebiges Programm entscheidet, ob Objekte zu einem bestimmten Zeitpunkt lebendig sind. Dieser müsste insbesondere entscheiden, dass der Teil eines Programms, in dem ein Objekt lebendig ist, terminiert. Ein solcher Algorithmus existiert jedoch nicht (vgl. [Sip13, Kap. 4.2]). □

Aus diesem Grund betrachten wir eine schwächere Eigenschaft von Objekten: die Erreichbarkeit über Referenzen. Dafür gehen wir von einer Menge **ROOTS** von **Basisobjekten** (engl. *root objects*) aus. Diese sind dadurch gekennzeichnet, dass der Mutator unmittelbaren Zugriff auf sie hat, ohne dafür zunächst ihre Adresse aus den Feldern anderer Objekte beschaffen zu müssen. Hierzu zählen zum Beispiel statische Objekte, deren Position im Speicher bereits zur Compilezeit bekannt ist, oder Objekte, die sich in *stack frames* befinden. Alle weiteren Objekte, die zur Laufzeit dynamisch erzeugt werden, gelten als erreichbar, wenn auf sie über eine Folge von Referenzen zugegriffen werden kann, wobei diese in den Feldern von Objekten gespeichert sind und die erste Referenz auf ein Basisobjekt verweist. Einfacher ausgedrückt: Ein Objekt ist erreichbar, wenn der Mutator die Möglichkeit hat, mittelbar oder unmittelbar über Referenzen auf das Objekt zugreifen zu können. Formal definieren wir diese Eigenschaft wie folgt:

definieren

Definition 1.3 (Erreichbarkeit):

Jedes Element der Menge \mathcal{R} der erreichbaren Objekte ist durch endlich häufige Anwendung der folgenden beiden Regeln konstruiert:

- (1) Ist $a \in \text{ROOTS}$, so folgt $a \in \mathcal{R}$.
- (2) Ist $a \in \mathcal{R}$, b ein weiteres Objekt und existiert ein $i \in \mathbb{N}$ mit $*a[i] = b$, so folgt $b \in \mathcal{R}$. In diesem Fall schreiben wir auch $a \rightarrow b$.

Diese Definition garantiert zwar nicht, dass jedes erreichbare Objekt auch lebendig ist. Davon ausgehend, dass unerreichbare Objekte auch nicht *wiedergefunden* werden können, können wir jedoch mit Sicherheit sagen, dass unerreichbare Objekte nicht mehr verwendet werden und gefahrlos durch den Kollektor freigegeben werden dürfen.

Die Erreichbarkeit von Objekten lässt sich mithilfe eines sogenannten **Objektgraphen** visualisieren. Jedes existierende Objekt korrespondiert dabei zu einem Knoten des Graphen. Besitzt ein Objekt a in mindestens einem seiner Felder eine Referenz auf ein weiteres Objekt b , so wird dies durch eine gerichtete Kante zwischen den entsprechenden Knoten dargestellt. Ein Objekt ist somit nicht erreichbar, wenn es im Objektgraphen keinen Pfad zu ihm gibt, der in einem Basisobjekt startet. Objektgraphen werden uns im Rahmen dieser Arbeit bei der Veranschaulichung der vorgestellten Algorithmen dienlich sein.

Definition 1.4 (Objektgraph):

Sei O eine Menge von Objekten. Ein gerichteter Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge $E \subseteq V \times V$ heißt Objektgraph, wenn eine bijektive Abbildung $\varphi: O \rightarrow V$ existiert, sodass für je zwei Objekte $a, b \in O$ gilt:

$$a \rightarrow b \iff (\varphi(a), \varphi(b)) \in E.$$

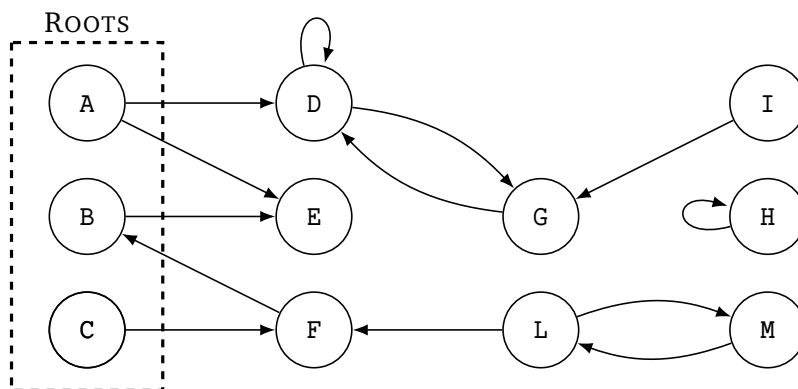


Abbildung 1.1.: Beispiel für einen Objektgraphen. Die Objekte A, B und C sind Basisobjekte. Die Objekte H, I, L und M sind in dieser Konstellation nicht erreichbar.

An dieser Stelle formulieren wir ein Korrektheitskriterium für Garbage-Collection-Algorithmen. Dieses besteht aus der Anforderung, dass keine noch benötigten Daten zerstört werden.

Definition 1.5 (Korrektheit für Garbage-Collection-Algorithmen):

Ein Garbage-Collection-Algorithmus ist *korrekt*, wenn er keine lebendigen Objekte freigibt.

Gemäß Definition 1.3 ist es folglich hinreichend zu zeigen, dass nur nicht erreichbare Objekte freigegeben werden, um Korrektheit nachzuweisen.

Man kann an dieser Stelle fragen, warum wir nicht voraussetzen, dass die Ausführung eines Garbage-Collection-Algorithmus die Freigabe *sämtlicher* nicht erreichbaren Objekte anfordert. Tatsächlich werden wir sehen, dass es aus Performancegründen vorteilhaft sein kann, nur einen Teil des nicht mehr benötigten zugewiesenen Speichers zu bereinigen, um längere Wartezeiten zu vermeiden. Ein solches Kriterium wäre daher zu restriktiv.

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  $\wedge$  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.

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. Somit terminiert der gesamte Algorithmus.

Weiter werden lediglich nicht markierte Objekte freigegeben (Zeile 4–6 in Algorithmus 2.2). Bleibt ein Objekt obj unmarkiert, so ist obj kein Basisobjekt, da $markStart$ alle Basisobjekte markiert. Somit wird kein Basisobjekt freigegeben. Wir zeigen, dass es nach der Markierungsphase zudem keine zwei Objekte a, b mit $a \rightarrow b$ gibt, sodass a markiert und b unmarkiert ist: Da a markiert ist, wurde a auch der Menge $todo$ hinzugefügt (Zeile 9f bzw. 18f in Algorithmus 2.1). Entsprechend gab es eine Iteration der **while**-Schleife mit $obj = a$. Gilt nun $a \rightarrow b$, so ist $\&b \in POINTERS(a)$. Folglich wird in einer Iteration der **for each**-Schleife mit $adr = \&b$ auch $\ast(\&b) = b$ markiert, falls b nicht schon zuvor markiert wurde. Es existiert somit keine Referenz von einem erreichbaren auf ein unmarkiertes Objekt, weswegen keine erreichbaren Objekte freigegeben werden. \square

Die Bedingung, dass der Mutator während des Markierens pausiert wird, ist tatsächlich notwendig, um zu 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:

```
1: B.ref  $\leftarrow$  &C  
2: A.ref  $\leftarrow$  null  
3: A.ref  $\leftarrow$  &C  
4: B.ref  $\leftarrow$  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.

Situationen, in denen die nebenläufige Ausführung von Programmsegmenten zu unvorhersehbarem Verhalten führt, werden als *race conditions* bezeichnet. Algorithmen, die zur Vermeidung von *race conditions* zwischen Kollektor und Mutator die Arbeit des letzteren unterbrechen, werden auch als *Stop-the-World-Algorithmen* (vgl. [JHM11, S. 17]) bezeichnet.

2.2 Markierungsmöglichkeiten

2.3 Drei-Farben-Abstraktion

Da das Anhalten des Mutators während eines gesamten Garbage-Collection-Zyklus zu vergleichsweise großen Verzögerungen führt, ist eine Optimierung der Markierungsphase zielführend. Wünschenswert ist, Kollektor und Mutator möglichst häufig eine nebenläufige Ausführung zu ermöglichen, ohne dabei die Korrektheit der Garbage Collection zu gefährden. Wir haben gesehen, dass die Manipulation von Referenzen während der Markierungsphase dazu führt, dass Verweise von markierten auf unmarkierte Objekte entstehen können, sodass erreichbare Objekte unmarkiert bleiben. Um dies zu vermeiden, könnte man als ersten Ansatz auf die Idee kommen, beim Schreiben einer neuen Referenz in ein Feld eines Objekts das Ziel dieser Referenz sofort zu markieren. Dies ist jedoch nur scheinbar eine Lösung: Da das Ziel ebenfalls Referenzen auf unmarkierte Objekte bereithalten könnte, entstehen dadurch möglicherweise neue Referenzen von markierten auf unmarkierte Objekte. Von Dijkstra et al. stammt ein Ansatz, der diese Idee aufgreift und um ein *Zwischenstadium* erweitert [Dij+78, S. 969f]. Diese ermöglicht es, markierte Objekte, die bereits komplett abgearbeitet wurden, von solchen zu unterscheiden, die bislang lediglich entdeckt wurden. Dazu werden Objekte mit drei verschiedenen Farben markiert:

weiß: Das Objekt wurde bislang nicht als erreichbar identifiziert. Bleibt es nach Ende der Markierungsphase weiß, kann es freigegeben werden.

grau: Das Objekt ist erreichbar, allerdings wurden die Felder des Objekts noch nicht auf Referenzen zu weiteren Objekten überprüft.

schwarz: Das Objekt ist erreichbar und alle Felder des Objekts wurden bereits überprüft.

Zu Beginn des Algorithmus sind alle existierenden Objekte weiß. Wir modifizieren den ursprünglichen Mark-and-Sweep-Algorithmus 2.1 so, dass Objekte bei ihrer Entdeckung grau und nach Abarbeitung ihrer Felder schwarz markiert werden. Hierfür existiert eine atomare Prozedur *setColor*, die die Markierung eines Objekts auf eine bestimmte Farbe WHITE, GRAY oder BLACK setzt. Auf diese Art bleiben nicht mehr erreichbare Objekte weiß und können anschließend als löscher identifiziert werden.

Analog wird die *sweep*-Prozedur in Algorithmus 2.2 so abgeändert, dass Objekte gelöscht werden, wenn sie weiß markiert sind. Andernfalls wird ihre Markierung zurück auf weiß gesetzt.

Algorithmus 2.3 Markierung mit Drei-Farben-Abstraktion (vgl. [Dij+78, S. 970])

Vorbedingung: Alle Objekte sind weiß markiert.

```

1: markStart():
2:   graySet  $\leftarrow \emptyset$                                 ▷ Grau markierte Objekte
3:   for each obj  $\in$  ROOTS
4:     if isWhite(obj)
5:       setColor(obj, GRAY)
6:       add(graySet, obj)
7:       mark()

8: mark():
9:   while graySet  $\neq \emptyset$ 
10:    obj  $\leftarrow$  remove(graySet)
11:    setColor(obj, BLACK)                                ▷ Objekt wird nun abgearbeitet
12:    for each adr  $\in$  POINTERS(obj)
13:      if (adr  $\neq$  null  $\wedge$  isWhite(*adr))
14:        setColor(*adr, GRAY)                            ▷ Referenzierte Objekte grau markieren
15:        add(graySet, *adr)

16: sweep():
17:   pos  $\leftarrow$  nextObject(HEAP_START)
18:   while pos  $\neq$  null
19:     if isWhite(*pos)
20:       free(pos)
21:     else setColor(*pos, WHITE)
22:     pos  $\leftarrow$  nextObject(pos)

```

Mithilfe dieser **Drei-Farben-Abstraktion** (engl. *tri-color abstraction*) können wir nun Referenzmanipulationen zulassen, die parallel zur Markierungsphase der Garbage Collection stattfinden: Wird in einem Objekt *a* eine Referenz auf ein Objekt *b* hinterlegt, so kann dies dazu führen, dass *b* erreichbar wird. Infolgedessen müssen auch alle von *b* referenzierten Objekte als erreichbar identifiziert werden. Somit ist es zielführend, *b* beim Setzen der Referenz grau zu markieren und zu graySet hinzuzufügen, sofern dies noch nicht der Fall ist oder die Felder von *b* bereits verfolgt wurden. Dies kann etwa mit einer *Schreibbarriere* (engl. *write barrier*) realisiert werden, die genau dann zum Einsatz kommt, wenn eine Referenz in ein Feld eines Objektes geschrieben wird (siehe Algorithmus 2.4). Auf diese Art kann es jedoch vorkommen, dass Objekte unerreichbar werden, nachdem sie bereits grau oder schwarz markiert wurden. Entsprechend verbleiben sie zunächst im Speicher und werden erst im nächsten Garbage-Collection-Zyklus entfernt.

Algorithmus 2.4 Schreibbarriere zur Manipulation von Referenzen in Objekten.

Input: Objekt *obj*, in dem Referenz gesetzt wird; Index des Feldes *i*; zu setzende Referenz *ref*

```
1: atomic writeRef(obj, i, ref):  
2:   if (ref  $\neq$  null  $\wedge$  isWhite(*ref))  
3:     setColor(*ref, GRAY)  
4:     add(grayList, *ref)  
5:   obj[i]  $\leftarrow$  ref
```

An dieser Stelle gehen wir auf die Terminierung und Korrektheit des modifizierten Mark-and-Sweep-Algorithmus ein.

Satz 2.2:

Der Mark-and-Sweep-Algorithmus mit Drei-Farben-Abstraktion terminiert und ist korrekt, sofern während der Arbeit des Kollektors keine neuen Objekte erzeugt werden.

Beweis: Zur Terminierung: Wir halten zunächst fest, dass Objekte während der Markierungsphase ausschließlich *dunkler* gefärbt werden: In Algorithmus 2.3 werden lediglich weiß markierte Objekte grau gefärbt (Zeile 5 und 14); eine Weißfärbung geschieht in dieser Phase grundsätzlich nicht. Analog zum ursprünglichen Algorithmus 2.1 enthält *graySet* nur grau markierte Objekte. Jedes Objekt befindet sich dadurch höchstens einmal in dieser Menge. Da *graySet* nach jeder Iteration der **while**-Schleife um ein Objekt reduziert wird und keine neuen Objekte erzeugt werden, terminieren auch hier **while**- und **for each**-Schleife nach endlich vielen Schritten. Zuletzt terminiert auch Bereinigungsphase (siehe Satz 2.1).

Zur Korrektheit: Erneut ist zu zeigen, dass keine Objekte unmarkiert bleiben, die erreichbar sind. Dies wäre genau dann der Fall, wenn nach der Markierungsphase ein schwarz markiertes Objekt eine Referenz auf ein weiß markiertes Objekt besitzt. Wir zeigen daher die Gültigkeit folgender Schleifeninvariante für die **while**-Schleife:

- (A) Es existieren keine zwei Objekte *a* und *b* mit $a \rightarrow b$, sodass *a* schwarz markiert und *b* weiß markiert ist.

Da zu Beginn laut Vorbedingung alle Objekte weiß markiert sind und bis zum Eintritt in die **while**-Schleife Objekte nur grau markiert werden, gilt (A) trivialerweise, da keine schwarz markierten Objekte existieren. Gelte nun (A) zu Beginn einer Schleifeniteration und sei *a* dasjenige Objekt, das der Menge *graySet* entnommen und schwarz gefärbt wird (Zeile 10f). Existiert nun ein weiteres Objekt *b* mit $a \rightarrow b$, so ist $\&b \in \text{POINTERS}(a)$. Ist *b* bereits grau oder schwarz markiert, so geschieht nichts. Andernfalls wird $\ast(\&b) = b$ grau markiert (Zeile 14). In beiden Fällen ist *b* am Ende der Schleife nicht weiß markiert. Da *a* das einzige Objekt ist, das in dieser Iteration

schwarz gefärbt wird, ist (A) nach der Iteration weiterhin gültig. Da die Schleife terminiert, die Invariante aufrecht erhalten bleibt und in der Bereinigungsphase nur weiß markierte Objekte freigegeben werden, ist der Algorithmus korrekt. \square

Die Atomizität der Schreibbarriere *writeRef* in Algorithmus 2.4 ist in der Tat notwendig, um die Korrektheit zu gewährleisten. Käme es unmittelbar nach der Markierung des Referenzziels, aber vor dem eigentlichen Setzen der Referenz in Zeile 5, zu einen kompletten Garbage-Collection-Zyklus, so würde die Markierung durch den Kollektor wieder aufgehoben. Damit bestünde im Anschluss die Möglichkeit, dass die Invariante (A) verletzt wird. Allerdings besteht die Möglichkeit, die Invariante (A) abzuschwächen und damit eine *feingranularere* Lösung zu ermöglichen. Für Details hierzu verweisen wir auf [Dij+78, S. 972ff]. Für einen Überblick über verschiedene Möglichkeiten von Lese- und Schreibbarrieren im Kontext der Drei-Farben-Abstraktion siehe zudem [Pir98].

Markierungsmöglichkeiten (Tricolor, Bitmapping); Lazy Sweeping

Teil II

Entwurf und Realisierung eines
Garbage-Collection-Simulators

Modellierung

3

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 den Seiten 4, 14–16, 18).
- [JHM11] Richard Jones, Antony Hosking und Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Boca Raton: CRC Press, 2011 (zitiert auf Seite 15).
- [JL96] Richard Jones und Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester: John Wiley & Sons, 1996 (zitiert auf den Seiten 2, 12, 13).
- [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 11).
- [Pir98] Pekka P. Pirinen. „Barrier Techniques for Incremental Tracing“. In: *SIGPLAN Notices* 34.3 (1998), S. 20–25 (zitiert auf Seite 18).
- [Sip13] Michael Sipser. *Introduction to the Theory of Computation*. 3. Aufl. Cengage Learning, 2013 (zitiert auf Seite 5).
- [TB14] Andrew S. Tanenbaum und Herbert Bos. *Modern Operating Systems*. 4. Aufl. Pearson, 2014 (zitiert auf Seite 3).
- [Wil92] Paul R. Wilson. „Uniprocessor Garbage Collection Techniques“. In: *IWMM '92* (1992), S. 1–42 (zitiert auf Seite 1).

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

Abbildungsverzeichnis

1.1	Beispiel für einen Objektgraphen	6
-----	--	---

Tabellenverzeichnis

Algorithmenverzeichnis

1.1	Methode <i>new</i> zur Erzeugung eines neuen Objekts	4
2.1	Naives Mark and Sweep – Markierung	12
2.2	Naives Mark and Sweep – Bereinigung	13
2.3	Markierung mit Drei-Farben-Abstraktion	16
2.4	Schreibbarriere zur Manipulation von Referenzen	17

Eigenständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Masterarbeit *Garbage-Collection-Algorithmen und ihre Simulation* 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)