Automatische Speicherverwaltung mit Regionen

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fredericiana zu Karlsruhe (TH)

genehmigte

Dissertation
von
Florian Liekweg
aus Pfullendorf

Tag der mündlichen Prüfung: 30. Januar 2007
Erster Gutachter: Prof. Dr. rer. nat. Dr. h. c. G. Goos
Zweiter Gutachter: Prof. Dr.-Ing. Bellosa

Danksagung

Ich bedanke mich bei allen Menschen,
die mir auf meine Fragen Antworten gegeben haben,
und bei allen Menschen,
auf deren Fragen ich keine Antwort wusste.

Inhaltsverzeichnis

1 Einführung **11**

2 Die These **13**

3 Umfeld und verwandte Arbeiten **17**

3.1 Echtzeitsysteme 17
3.1.1 Weiche Echtzeitsysteme 17
3.1.2 Harte Echtzeitsysteme 18
3.2 Speicherorganisation 20
3.2.1 Übersicht 20
3.2.2 Organisation des Hauptspeichers 22
3.3 Freispeicherverwaltung 24
3.4 Explizite Speicherverwaltung 27
3.5 Automatische Speicherbereinigung 30
3.5.1 Funktionsweise 30
3.5.2 Umsetzung 34
3.5.3 Inkrementelle Speicherbereinigung 36
3.6 Automatische Speicherverwaltung in funktionalen Programmen . . 40
3.7 Referenzzähler 42
3.8 Übersetzer-Optimierungen 46
3.9 Zusammenfassung der verwandten Arbeiten 48

4 Ansatz und Vorgehen **51**

4.1 Zielsetzung 51
4.2 Korrekter Umgang mit Haldenspeicher 52
4.3 Programmrepräsentation 54
4.3.1 Übersetzerstruktur 54
4.3.2 Statische Einmalzuweisung 55
4.3.3 Statische Programmstruktur 58
4.3.4 Firm-Graphen 62

4.3.5 Speicheranordnung und Polymorphie 69
4.4 Zeigeranalyse 71
4.4.1 Eigenschaften unserer Zeigeranalyse 72
4.4.2 Transferfunktionen 74
4.4.3 Konstruktion des Aufrufgraphen 76
4.4.4 Steuerung der Analysereihenfolge 77
4.5 Bestimmung der Lebenszeiten von Objekten 78
4.5.1 Zugriffsanalyse 78
4.5.2 Analysekontexte 79
4.5.3 Lebenszeiten von Objekten 80
4.6 Allokation in Regionen 84
4.7 Regionen 84
4.8 Konflikte zwischen Regionen 86
4.8.1 Entstehen von Konflikten 86
4.8.2 Auflösen von Konflikten 87
4.9 Zusammenfassung 90

5 Ergebnisse **93**
5.1 Einführung 93
5.2 Auswahl der Benchmarks 93
5.3 Quantifizierung der Analyse- und Transformationsergebnisse . . . 95
5.4 Durchführung der Laufzeitversuche 96
5.4.1 Erstellung der Benchmarks 96
5.4.2 Laufzeitbedarf 98
5.4.3 Speicherverbrauch 99

6 Rückschau und Ausblick **111**
6.1 Rückschau 111
6.2 Kritik 112
6.3 Offene Fragen 114
6.3.1 Kombination mit Speicherbereinigern 114
6.3.2 Deterministische Ausführung von Destruktoren 115

Abbildungsverzeichnis

3.1 Speicherhierarchie 21
3.2 Freilisten 25
3.3 Beispiel für Externe Fragmentierung 26
3.4 Lebenszyklus eines Objektes 28
3.5 Drei-Farben-Invariante 33
3.6 Notwendige Aktualisierungen von Referenzzählern bei einer Zuweisung 42
3.7 Beispiel für kaskadierende Löschung bei Referenzzählern 44

4.1 Struktur des Übersetzers 55
4.2 Beispiel für einen *Firm*-Graphen 68
4.3 Implementierung von *fat pointers* und normalen Referenzen 70
4.4 Aufrufgraph zu Beispiel 4.14 83
4.5 Programmcode zu Beispiel 4.15 87
4.6 Programmausführung zu Beispiel 4.15. 88

5.1 Speicherverbrauch von `BH` 106
5.2 Speicherverbrauch von `BiSort` 106
5.3 Speicherverbrauch von `Em3d` 107
5.4 Speicherverbrauch von `Health` 107
5.5 Speicherverbrauch von `MST` 108
5.6 Speicherverbrauch von `Perimeter` 108
5.7 Speicherverbrauch von `Power` 109
5.8 Speicherverbrauch von `TreeAdd` 109
5.9 Speicherverbrauch von `TSP` 110
5.10 Speicherverbrauch von `Voronoi` 110

Tabellenverzeichnis

4.1 Abhängigkeiten zwischen Haldenoperationen 57
4.2 Beispiele für Namensschemata 73

5.1 Umfang der Benchmarks 94
5.2 Allokations- und Regionenanweisungen 96
5.3 Parameter der Benchmarks 98
5.4 Laufzeiten der Benchmarks 99
5.5 Speicherbedarf und durchschnittl. Objektgrössen der Benchmarks 100
5.6 Allokationsraten der Benchmarks 100
5.7 Verhalten des Speicherbereinigers 102
5.8 Verhalten der Regionen 102
5.9 Speicherbelegung im Vergleich zum angeforderten Speicher 103

Kapitel 1

Einführung

„Why is this area so prone to folk wisdoms?“
Hans-P. Boehm

Dynamischer Speicher ist seit geraumer Zeit eine Eigenschaft moderner und verbreiteter Programmiersprachen und -systeme. Der Umgang mit dynamisch alloziertem Speicher ist jedem Nutzer von Computersystemen seit den ’60-er Jahren wohlbekannt. Dabei überdeckt der Begriff des „Nutzers“ hier das gesamte Spektrum vom Hobby-Programmierer bis zum professionellen Software-Entwickler [Spo04] oder zum Repräsentanten des Idealbildes — manche würden sagen, des Zerrbildes — des *Hackers*.

Dynamischer Speicher ist eine der vielen unterschiedlichen endlichen Ressourcen, die ein Programm zur Laufzeit verwalten muss. Genauso wie Deskriptoren für Dateien, wie offene Netzwerkverbindungen oder Fenster, die innerhalb einer graphischen Nutzungsoberfläche angezeigt werden, ist der Vorrat an dynamischem Speicher nach oben begrenzt. Werden endliche Ressourcen nicht mehr benötigt, müssen sie freigegeben werden, um zu vermeiden, dass der Programmlauf wegen Ressourcenmangel abgebrochen wird.

Dynamischer Speicher ist jedoch gerade die Ressource, die in Programmen am häufigsten angefordert wird, die die geringste Lokalität hat, und die am häufigsten die Grenzen zwischen Struktureinheiten des Programmes — Module, Komponenten, Klassen etc. — überschreitet. Der korrekte Umgang mit dynamischem Speicher — insbesondere das Aufspüren und Beheben von Fehlern dabei — ist damit die häufigste Aktivität innerhalb der Softwareentwicklung, die die Programmierer daran hindert, die Software inhaltlich weiter zu entwickeln.

Bei allem Fleiß, der in diese Aufgabe gesteckt wird: So manches Softwareprodukt, das die Ressource „dynamischer Speicher“ verwaltet, zeigt nach längerer Laufzeit, dass die Implementierung der Speicherverwaltung noch verbesserungswürdig ist. Es wird Speicher freigegeben, obwohl er noch genutzt wird, oder Speicher,

der nicht mehr benötigt wird, bleibt alloziert. Der Nutzer des Produktes sieht es entweder „abstürzen“, oder er stellt fest, dass nach und nach der Speicher seines Rechners „volläuft“, auch ohne dass die Arbeitslast des Produktes steigt. In jedem Falle ist ein Neustart des Programms nötig, sei es nach dem zwangsweisen oder freiwilligen Ende des letzten Laufes.

In den Systemen, die für sicherheitskritische Einsatzgebiete geschaffen werden, in lange laufenden, eingebetteten Systemen, machen die oben genannten Schwierigkeiten die Verwendung von dynamischem Speicher an sich problematisch.

In Anbetracht der möglichen Folgen, die eine Fehlfunktion dieser Systeme haben könnte, ist man bestrebt, über die Korrektheit solcher Systeme eine formelle, beweisbare Aussage machen zu können. Die Abwesenheit von Ursachen, die zu fehlerhafter Nutzung von dynamischen Speicher führen können, lässt sich jedoch nur schwer beweisen, selbst in einem nur informellen Sinne.

Einen Ausweg versprechen automatische Speicherverwaltungen, namentlich Speicherbereiniger [Wil90, BW88]. Alleine schon durch ihre Konstruktion verhindern Speicherbereiniger zwar nicht alle, jedoch viele Probleme, die bei der Speicherverwaltung auftreten können.

Bei Systemen, die Echtzeit-Bedingungen erfüllen, wird die Laufzeit von Speicherbereinigern, genauer, die mangelnde Genauigkeit, mit der sie sich vorhersagen lässt, zum Problem. Diejenigen (wenigen) Speicherbereiniger [Bak92, CB01, BCR03], die sich auch noch in Echtzeitsystemen einsetzen lassen, fallen durch hohe Kosten im Bereich der Laufzeit und des Speicherbedarfs auf [Det04].

Wir stellen in dieser Arbeit einen neuen Ansatz vor, dessen Konstruktion wir so gewählt haben, dass Vorteile von herkömmlicher Speicherverwaltung und von Speicherbereinigung kombiniert werden. Unser Ansatz liefert vollautomatisch eine korrekte, effiziente Speicherverwaltung.

Wir stellen in Kapitel 2 die Eigenschaften vor, die wir erreichen. In Kapitel 3 stellen wir das Umfeld dieser Arbeit im Detail vor, und nennen und beurteilen existierende Arbeiten auf unserem Gebiet. In Kapitel 4 stellen wir unseren Ansatz vor, und in Kapitel 5 präsentieren wir die Ergebnisse, die wir mit unserem Ansatz erzielt haben. In Kapitel 6 rekapitulieren wir unser Vorgehen und die Resultate und gehen auf offene Fragen ein.

Kapitel 2

Die These

„Multum, non multa."
Plinius d. J., Epistulæ

In dieser Arbeit stellen wir eine neuartige Methode vor, mit der der Haldenspeicher in Programmen, die in einer imperativen Programmiersprache geschrieben sind, automatisch verwaltet werden kann. Im Gegensatz zu bestehenden Ansätzen ist es weder erforderlich, dass der Programmierer explizite Anweisungen zum Löschen von Objekten in das Programm einfügt, noch ist es nötig, dass zur Laufzeit ein Speicherbereinigungsmechanismus [Jon99] den Haldenspeicher bereinigt.

Die Ergebnisse unserer Methode erfüllen eine Reihe von Kriterien, die von existierenden Ansätzen nicht oder nicht vollständig erfüllt werden können, oder deren Erfüllung zwar von Fall zu Fall erreicht, aber nicht garantiert werden kann:

Neuheit

Unsere Methode stellt eine Neuheit auf ihrem Gebiet dar. Andere Methoden ([HHN94, BC92, und andere]) verlangen manuelle Arbeit des Programmierers, können sich ([GS00a, und andere]) nicht um alle Objekte kümmern, verlassen sich auf Mechanismen, die zur Laufzeit arbeiten ([BW88, Han90, Jon99, Hir04, und andere]), oder sind nur für funktionale Sprachen geeignet ([GA01, NKY04, und andere]). Unsere Methode ist die erste, mit der durch den Übersetzer ein Programm, das in einer imperativen Sprache geschrieben ist, so instrumentiert wird, dass vollautomatisch alle zur Laufzeit allozierten Objekte gelöscht werden können.

Universalität

Unsere Methode ist universell. Sie lässt sich auf jede imperative Programmiersprache ansetzen. Sie verlangt nach keinerlei Einschränkungen der Ausdrucksmächtig-

keit der Sprache wie [DKAL05], auf der sie eingesetzt wird, und sie verlangt keine Erweiterung der Syntax wie [HHN94].

Vollständigkeit

Unsere Methode ist vollständig: Jedes Objekt, das auf der Halde alloziert wird, wird garantiert wieder gelöscht, wenn alle Zugriffe auf dieses Objekt stattgefunden haben. Wir beschränken uns nicht darauf, nur einen Teil der Objekte einzufangen, wie [TT93, Bla99, Tra00, und andere], wobei die verbleibenden Objekte durch einen zusätzliche Speicherbereinigungsmechanismus behandelt werden müssen.

Sicherheit

Unsere Methode bietet die Sicherheit, dass kein Objekt dealloziert wird, bevor nicht der letzte Zugriff auf das Objekt stattgefunden hat. Es ist nicht möglich, dass ein Objekt vorzeitig dealloziert wird. Unsere Methode garantiert auch, dass jedes Objekt, das alloziert wird, genau einmal gelöscht wird. Es ist nicht möglich, dass der Speicher eines Objektes der Speicherverwaltung mehr als einmal als frei gemeldet wird.

Analysierbarkeit und Vorhersagbarkeit

Für die Deallokation von Objekten ist ein gewisser Laufzeitaufwand notwendig. Bis Objekte gelöscht werden, ist ihr Speicher belegt und kann nicht für andere Allokationen verwendet werden.

Durch unsere Methode bestimmen wir, wie lange Objekte alloziert bleiben müssen. Wir fügen Anweisungen in das Programm ein, die Objekte löschen. Damit geben wir auch an, an welcher Stelle des Programms — und damit zu welchem Zeitpunkt der Programmausführung — der Laufzeitaufwand für das Löschen der Objekte fällig wird, wie hoch er ist, und welche Objekte mit dieser Anweisung gelöscht werden.

Für ein Programm, dessen präzises Laufzeitverhalten von Interesse ist — insbesondere für ein Programm, das Echtzeit-Bedingungen erfüllen muss — lässt sich mit unserer Methode die Laufzeit deshalb präzise analysieren. Im Gegensatz dazu muss man bei herkömmlichen Ansätzen auf konservative Abschätzungen zurückgreifen [Bak78, LH83, Det04].

Speicher- und Laufzeiteffizienz

Die mit unserer Methode übersetzten Programme sind, sowohl was den die Laufzeit angeht, effizient. Sie sind den Programmen, die mit anderen Methoden der Speicherverwaltung arbeiten, ebenbürtig. Man muss keinen systematischen Preis in Form von erhöhtem Laufzeitbedarf bezahlen [BCR03]. Auch was den Speicherbedarf angeht, können wir in vielen Fällen mit anderen Methoden der Speicherverwaltung mithalten.

Kapitel 3

Umfeld und verwandte Arbeiten

In diesem Kapitel stellen wir das Umfeld dieser Arbeit vor, und besprechen existierende Ansätze zur Speicherverwaltung.

3.1 Echtzeitsysteme

Bedeutungen des Begriffs „Echtzeit“ Mit dem Begriff der „Echtzeit“ und der „Echtzeitsysteme“ werden unterschiedliche Bedeutungen verbunden. Um Missverständnisse zu vermeiden, stellen wir im Folgenden die möglichen Bedeutungen vor. Was die „weichen Echtzeitsysteme“ aus Abschnitt 3.1.1 angeht, wird es bei dieser Vorstellung bleiben; die „harten Echtzeitsysteme“ aus Abschnitt 3.1.2 bilden die Grundlage unserer weiteren Argumentation in dieser Arbeit.

3.1.1 Weiche Echtzeitsysteme

Unter dem Begriff der „weichen Echtzeitsysteme“ (engl: *soft real-time systems*) fasst man solche Systeme zusammen, an deren Reaktionszeit und Durchsatz gewisse Anforderungen gestellt werden. Häufig wird verlangt, dass das System eine gewisse Last an Eingaben bearbeiten kann, ohne mit der Bearbeitung einzelner Aufträge wesentlich in Verzug zu geraten.

Die Eignung eines solchen Systems für einen gegebenen Einsatz ist nicht allein durch Inspektion der verwendeten Software zu beurteilen, sondern kann nur mit Expertenwissen über die Systemumgebung und die typische Arbeitslast geschehen. Auch wenn diese Systeme in der Fachliteratur häufig einfach als „Echtzeitsysteme“ bezeichnet werden, sind sie streng von den im nächsten Abschnitt beschriebenen Echtzeitsystemen zu unterscheiden.

3.1.2 Harte Echtzeitsysteme

Echtzeitsysteme werden zur Unterscheidung von den o.g. „weichen Echtzeitsystemen“ manchmal als “harte Echtzeitsysteme“ (engl: *hard real-time systems*) bezeichnet.

Definition von Echtzeitsystemen Die Anforderungen an ein Echtzeitsystem verlangen nicht nur, dass aus den Eingaben gemäss einer Spezifikation korrekt Ausgaben berechnet werden, sondern dass diese Ausgabe auch innerhalb eines festgelegten Zeitraumes fertig berechnet vorliegt. In der Literatur findet sich zwar keine einheitliche Definition für Echtzeitsysteme; jedoch unterscheiden sich die existierenden Definitionen nicht wesentlich. Wir geben in Definition 3.1 eine weithin akzeptierte Version an:

Definition 3.1 (Echtzeitsystem) *Ein Echtzeitsystem ist ein Computersystem, in dem die Korrektheit des Systems nicht nur vom logischen Ergebnis der Berechnung abhängt, sondern auch vom physikalischen Moment, in dem das Ergebnis produziert wird.*

nach Hermann Kopetz, TU Wien

Umgangssprachlich ausgedrückt wird bei einem Echtzeitsystem garantiert, dass nach einer Eingabe das Ergebnis geliefert wird, bevor eine bestimmte Zeitspanne abgelaufen ist.

Dieses System kann in einer Umgebung betrieben werden, die sich darauf verlässt, dass die Zeitschranken eingehalten werden, indem Ergebnisse immer rechtzeitig geliefert werden. Dies kann zum Beispiel bedeuten, dass auf eine Unterbrechung innerhalb einer vorgegebenen Zeit reagiert wird, oder dass eine wiederholt auszuführende Aufgabe mit einer vorgegebenen Frequenz ausgeführt wird.

Echtzeit-Analyse Wann ist ein Software-System, installiert auf einer Hardwareplattform, ein Echtzeitsystem im Sinne von Definition 3.1? Um diese Frage zu beantworten, muss die Laufzeit der Software analysiert werden. Interessant ist hierbei die Laufzeit all der Ausführungspfade, die in der Berechnung des Ergebnisses beschritten werden können. Dabei muss für jede Schleife die maximale Anzahl an Iterationen und für jede Rekursion die maximale Rekursionstiefe bekannt sein. Für Anweisungen, die Interaktionen mit der Hardware auslösen, muss die maximale Ausführungszeit der Hardwareaktion berücksichtigt werden. Bei Anweisungen, die nicht in der Anwendung selber implementiert sind, sondern die von der Laufzeitumgebung oder dem Betriebssystem zur Verfügung gestellt werden, muss deren maximale Laufzeit ermittelt werden. In der englischsprachigen Literatur nennt man

die maximale Laufzeit häufig *worst case exectuion time (WCET)*. Auf Operationen, die mit der Freispeicherverwaltung zu tun haben, werden wir in Abschnitt 3.2 noch zu sprechen kommen.

Auslastung von Echtzeitsystemen Um ein Softwaresystem auf einer Hardware-Platform so betreiben zu können, dass dieses System gegebene Anforderungen an die Leistungsfähigkeit erfüllt, muss die Hardware-Platform ausreichend dimensioniert werden: Der Prozessor muss ausreichend schnell sein, der Hauptspeicher muss gross genug sein, die Datenübertragungsraten zwischen den Komonenten müssen gross genug sein. Um sicher zu stellen, dass die Software für ein Echtzeitsystem die an sie gestellten Echtzeitanforderungen erfüllt, muss die Hardware-Platform so grosszügig ausgelegt sein, dass die Software ihre Berechnungen schnell genug ausführen kann, so dass sie die Ergebnisse rechtzeitig liefern kann. Dabei muss man sich an der maximalen Laufzeit orientieren, die in der Echtzeit-Analyse ermittelt wurde. Die *durchschnittliche* Laufzeit, die tatsächlich anfällt, liegt naturgemäss unter der maximalen Laufzeit. Die Differenz zwischen durchschnittlicher und maximaler Laufzeit verbringt das System im Leerlauf, ohne die Hardware zu nutzen.

In der Praxis stellt sich die Frage, wie groß die Differenz zwischen maximaler und durchschnittlicher Laufzeit ist. Liegt die durchschnittliche Laufzeit nämlich sehr weit unter der maximalen Laufzeit, die wir in der Echtzeitanalyse erhalten haben, so kann das Rechensystem ständig nur mit einer sehr geringen Auslastung betrieben werden. Damit ergibt sich ein sehr schlechtes Verhältnis zwischen der Leistungsfähigkeit der Hardware, die auch die Kosten der Hardware bestimmt, und dem Grad, in dem diese Leistung ausgenutzt wird. Wir zeigen in Abschnitt 3.5, wie diese Situation bei der Verwendung von automatischen Speicherbereinigern entstehen kann.

Ressourcen-Verbrauch Um sicherzustellen, dass ein Echtzeitsystem zur Laufzeit nicht aufgrund mangelnder Ressourcen abgebrochen wird, muss der Bedarf an Ressourcen durch das System bekannt sein. Für diese Arbeit ist im Folgenden nur eine einzige Ressource interessant, nämlich die Ressource „Freispeicher“. Der Verbrauch an Freispeicher durch das System muss so genau bekannt sein, dass ein Abbruch wegen Speichermangels ausgeschlossen werden kann.

Da die Laufzeit von Programmteilen, die Datenstrukturen anlegen, traversieren und ändern, bekannt ist, hat man damit auch Informationen über den möglichen Umfang der Datenstrukturen.

Umfang von Datenstrukturen Die Analyse der Grösse von Datenstrukturen und damit des Verbrauchs an Haldenspeicher gestaltet sich am einfachsten, wenn man die Grössen von Datenstrukturen durch Konstanten begrenzt. Damit wird auch

gleichzeitig der Laufzeit von Algorithmen, die diese Datenstrukturen traversieren, eine feste obere Schranke gesetzt. Durch dieses Vorgehen wird jedoch auch die Ausdrucksmächtigkeit des Programmierers stark eingeschränkt.

Lässt man jedoch zu, dass sich die Grösse der Datenstrukturen dynamisch ändert, so muss man jederzeit für jede Datenstruktur über ihren aktuellen Umfang Bescheid wissen, damit man gesicherte Aussagen über den aktuellen Speicherbedarf für die Datenstrukturen und über den Laufzeitbedarf für die Traversierung der Datenstrukturen machen kann.

Hieraus ergibt sich allerdings nicht unmittelbar, welche Blöcke des Freispeichers an welchen Stellen der Programmausführung *nicht mehr* benötigt werden. Diesem Problem widmen wir uns in den folgenden Abschnitten.

3.2 Speicherorganisation

3.2.1 Übersicht

In Abbildung 3.1 ist die Speicherhierarchie beschrieben, wie sie bei heute üblichen Rechensystemen zu finden ist. Wir werden die unterschiedlichen Speicherarten im Folgenden vorstellen. Wie in Abbildung 3.1 gezeigt, unterscheiden sich die Speicherarten in Latenz (wie lange dauert es, bis eine Anfrage bearbeitet wird?), in der Zugriffszeit (wie lange dauert es, bis die Anfrage bearbeitet ist?). Sie unterscheiden sich auch in den Kosten pro Speichereinheit. Nicht überraschend ist, dass mit fallender Latenz und kürzeren Zugriffszeiten die wirtschaftlichen Kosten pro Speichereinheit steigen. Deshalb findet man in Rechnern eine ausgewogene, balancierte Kombination [HP01] aus kleinen, schnellen Speichern und umfangreichen Speichern, die höhere Latenz- und Zugriffszeiten haben.

Register Die Nutzung der Register wird durch den Register-Allokator [Boe05, Hac06] des Übersetzers bestimmt. Die Register sind auf dem Prozessor untergebracht. Auf ein Register wird als Teil der Ausführung eines Maschinenbefehls zugegriffen. Die Zugriffszeit beträgt also weniger als die Ausführungszeit des Befehls.

Cache-Speicher Die Nutzung des Prozessor-Cachespeichers wird durch Strategien [HS89] bestimmt, die im Cache in Hardware implementiert sind. Cache-Speicher ist ein assoziativer Speicher, dessen Aufgabe es ist, den wiederholten Zugriff auf häufig genutzte Einträge des Hauptspeichers zu beschleunigen. In einem Rechner kann ein einziger Cache zwischen Prozessor und Hauptspeicher geschaltet werden, oder es kann aus mehreren Caches eine Hierarchie aufgebaut werden. In Abbildung 3.1 wird eine zweistufige Hierarchie gezeigt. Cache-Speicher kann auf dem

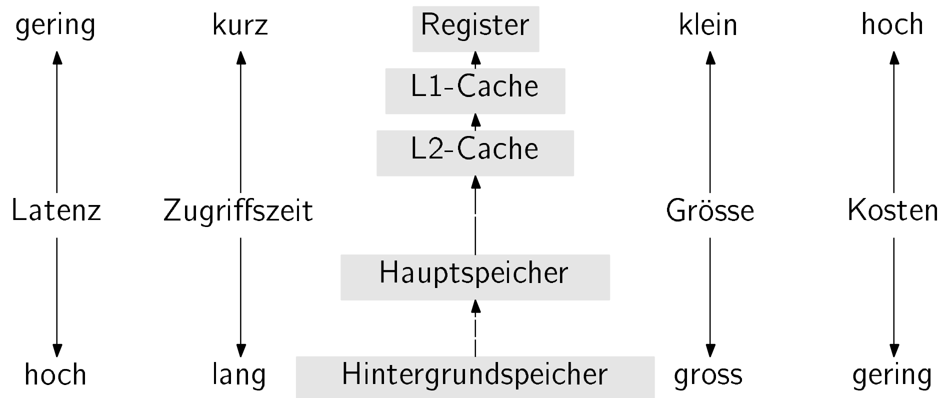


Abbildung 3.1: Speicherhierarchie

Prozessor untergebracht werden, was für kürzere Latenz- und Zugriffszeiten sorgt, während es aber auch die Kosten erhöht; oder er kann ausserhalb des Prozessors untergebracht werden, wodurch geringere Kosten verursacht werden und eine höhere Kapazität ermöglicht wird, was aber auch höhere Latenz- und Zugriffszeiten verursacht. Der Zugriff auf Daten, die im Cache-Speicher vorhanden sind, kostet wenige Prozessorzyklen. Zum Beispiel vergehen beim *Itanium*-Prozessor der Firma *Intel* zwischen drei und fünf Zyklen, bis ein Datenwert aus dem L1-Cache geliefert wird; zwischen neun und zehn Zyklen vergehen, bis der L2-Cache einen Wert liefert.

Hauptspeicher Im Hauptspeicher des Rechners kann jeder Prozess Speicherbereiche anfordern. Diese Bereiche stehen dem Prozess in den in Abschnitt 3.3 genannten Arten zur Verfügung. Hauptspeicher wird — von einigen speziellen Prozessoren abgesehen — ausserhalb des Prozessors untergebracht. Der Zugriff auf den Hauptspeicher kostet einige hundert Prozessorzyklen. Der Itanium-Prozessor, den wir im vorigen Abschnitt schon erwähnt haben, muß — je nach Auslegung des Speichersystems — zwischen 150 und 300 Zyklen warten, bis Daten aus dem Hautspeicher geliefert werden.

Hintergrundspeicher wird von der virtuellen Speicherverwaltung des Betriebssystems genutzt, um Bereiche des Hauptspeichers auszulagern. Der Hintergrundspeicher wird durch Massenspeichermedien, wie Festplatten oder seltener durch Bandlaufwerke implementiert. Bei diesen Medien bewegt sich die Zugriffszeit im Bereich von wenigen Millisekunden (bei Festplatten) bis zu mehreren Sekunden (bei Bandlaufwerken). Die Anzahl der Zyklen, die ein Prozessor in dieser Zeit ausführen kann, geht in die Tausende.

3.2.2 Organisation des Hauptspeichers

Der physikalische Speicher, der in einem Rechner vorhanden ist, steht einem Anwendungsprogramm nicht direkt zur Verfügung, sondern wird dem Programm durch das Betriebssystem zugeteilt und durch Routinen der Laufzeitumgebung verwaltet. Dem Programm steht der Hauptspeicher als Haldenspeicher zur Verfügung, wie es am Ende dieses Abschnittes erklärt wird.

Anweisungs- und Datenspeicher Zunächst ist der Programmcode des Programms in einem Teil des Speichers abgelegt, der vom Betriebssystem zugeteilt wird. Dieser Speicher wird beim Start des Programms durch das Betriebssystem zur Verfügung gestellt und mit den Anweisungen des Programms beschrieben. Zwar kann es vorkommen, dass der Inhalt dieses Speichers zur Laufzeit geändert wird, zum Beispiel weil das Programm dynamisch neuen Code nachlädt, jedoch besteht keine Möglichkeit, innerhalb dieses Speichers Daten des Programmlaufs abzulegen. Wir wenden uns deshalb dem Teil des Speichers zu, innerhalb dessen das Programm zur Laufzeit Daten ablegt, dem Hauptspeicher.

Der Hauptspeicher, der einem Prozess zugeteilt wird, steht diesem Prozess in drei Bereichen zur Verfügung:

Statischer Speicher Der statische Speicher (engl. *static segment*) hat eine feste Grösse, die durch das Programm bestimmt wird. In dem statischen Speicher werden Laufzeitkonstanten und globale Variablen abgelegt.

Aufrufstapel Auf dem Aufrufstapel werden alle Daten abgelegt, die für die Ausführung einer Prozedur des Programmes nötig sind. Um die Daten für einen neuen Prozeduraufruf abzulegen, wird eine neue Aufrufschachtel auf den Stapel gelegt. Diese Aufrufschachtel wird nach Ende der Prozedurausführung wieder abgeräumt. Innerhalb der Aufrufschachtel werden Verwaltungsdaten für den Prozeduraufruf, wie die Rücksprungadresse und Zeiger auf die dynamische Umgebung gespeichert; zusätzlich werden auf der Aufrufschachtel alle prozedurlokalen Variablen untergebracht, und alle Zwischenergebnisse, die bei der Berechnung von Ausdrücken im Prozessor nicht in dessen Registern gehalten werden können.

Nachdem eine Aufrufschachtel am Ende eines Prozeduraufrufes abgebaut wurde, verlieren alle in ihr enthaltenen Variablen ihre Gültigkeit. Dies ist so zu verstehen, dass nach dem Abbau der Aufrufschachtel nicht mehr garantiert ist, dass die dort bisher abgelegten Variablen ihren Wert beibehalten. Solange innerhalb des Programmes über lokale Variable auf diese Daten zugegriffen wird, ist dies bedeutungslos, da diese Variablen nach Prozedurende nicht mehr sichtbar sind. In manchen Programmiersprachen, wie zum Beispiel in `C`, ist es jedoch möglich, die Speicheradresse

einer Variablen zu erhalten; wird diese Speicheradresse über die Laufzeit eines Prozeduraufrufes hinausgetragen, so kann eine Dereferenzierung dieser Adresse unvorhersagbare Resultate liefern, da die Aufrufschachtel, die die referenzierte Variable enthielt, abgebaut wurde.

Haldenspeicher In beinahe jeder modernen imperativen Programmiersprache steht mit dem Haldenspeicher ein sehr vielseitig verwendbarer Speichertyp zur Verfügung. Dem Prozess steht hier zunächst ein linearer Speicherbereich zur Verfügung. Die Grösse dieses Speicherbereiches wird zum Start des Prozesses durch einen Vorgabewert (*default*) bestimmt. Dieser Vorgabewert ist typischerweise ein ganzzahliges Vielfaches der Grösse der Speicherseiten (engl. *memory page*). Durch geeignete Aufrufe an das Betriebssystem (unter UNIX-Varianten zum Beispiel durch den `brk(2)`-Aufruf) kann die obere Grenze dieses Bereichs verschoben werden.

Die Obergrenze wird nicht in variablen Inkrementen verschoben, sondern gleich um die Grösse einer Speicherseite erhöht.

Vergrösserung der Halde Allen in der Praxis anzutreffenden Verwaltungsmechanismen für den Haldenspeicher — insbesondere die, die wir im kommenden Abschnitt vorstellen — ist es gemein, dass sie den Haldenspeicher zwar nötigenfalls vergrössern, ihn jedoch nie verkleinern. Dies hat zwei Gründe.

Der erste Grund ist, dass man — in Ermangelung besserer Informationsquellen — das aktuelle Verhalten des Programms als eine Schätzung für das zukünftige Verhalten nimmt. Man unterstellt also, dass es in Zukunft ähnlich viel Speicher brauchen wird, so dass eine Verkleinerung der Halde zum aktuellen Zeitpunkt nur eine Vergrösserung zu einem späteren Zeitpunkt nötig machen würde.

Der zweite Grund besteht darin, dass, wenn nur ein einziger Bereich einer Seite noch vom Programm belegt ist, die Halde nicht unter diese Seite verkleinert werden kann. Die Chancen, die Halde jemals verkleinern zu können, werden als so schlecht eingeschätzt, dass man den Laufzeitaufwand für eine Überprüfung, ob man die Halde verkleinern könnte, im Verhältnis dazu als zu hoch einstuft.

Verwaltung durch die Laufzeitumgebung Der Haldenspeicher wird durch die Laufzeitumgebung in einer höheren, abstrakteren Form präsentiert. Auf diese Abstraktion werden wir im kommenden Abschnitt 3.3 eingehen.

3.3 Freispeicherverwaltung

Den vorigen Abschnitt hatten wir mit der Beschreibung des Haldenspeichers als linearen Speicherbereich beendet, der sich durch geeignete Systemaufrufe vergrössern lässt. Die Aufgabe der Freispeicherverwaltung in der Laufzeitumgebung ist es, diesen linearen Speicherbereich so einzuteilen, dass dem Programm — geschrieben in einer höheren Programmiersprache — eine geeignete Abstraktion zur Verfügung steht.

Innerhalb eines Programmes wird Speicher in Form von Speicherblöcken verschiedener Grösse benötigt. Solche Blöcke muss die Freispeicherverwaltung dem Programm zur Verfügung stellen. Auf Anfrage wird dem Programm ein Block der verlangten Grösse durch Angabe der Speicheradresse geliefert, an der der Block beginnt (typischerweise die zahlenmässig kleinste Adresse innerhalb des Blockes). Diese Anfrage eines Programmes, und die Aktionen, die die Freispeicherverwaltung unternimmt, um sie zu beantworten, nennen wir *Allokation*.

Die Freispeicherverwaltung löst nur das Problem, den linearen Haldenspeicher in Blöcke aufzuteilen, innerhalb derer ein Anwendungsprogramm seine Datenstrukturen ablegen kann; die Frage, wie die damit geschaffene Ressource „Freispeicher“ verwaltet werden soll, wird hier nicht behandelt. Wir gehen in den Abschnitten 3.4 ff. darauf ein. Gegenüber dem Programm, das direkt mit der Freispeicherverwaltung interagiert, erwarten wir, dass aller Speicher, der von der Freispeicherverwaltung angefordert wird, auch wieder an diese zurückgegeben wird. Die Operation, mit der ein Block an die Freispeicherverwaltung zurückgegeben wird, nennen wir *Deallokation*.

Freispeicherlisten Eine sehr häufig verwendete Verwaltungsstruktur für den Freispeicher ist die der Freispeicherlisten [Com64]. Der Freispeicher wird in Blöcke vorgegebener Grössen aufgeteilt; Blöcke gleicher Grösse werden in einer Liste (meistens in einer verketteten Liste) zusammengefasst. Eine Anfrage des Programms nach einem Block wird dadurch beantwortet, dass die verlangte Grösse auf die nächstgrössere Blockgrösse aufgerundet wird, und ein Block dieser Blockgrösse aus seiner Liste ausgehängt und dem Programm übergeben wird. Werden Blöcke freigegeben, so werden sie in die Liste wieder eingehängt. In Abbildung 3.2 ist dies beispielhaft dargestellt.

Interne Fragmentierung Die Differenz zwischen verlangter Blockgrösse und der tatsächlichen Blockgrösse wird im Freispeicher zwar belegt, aber vom Programm nicht genutzt. Die Summe allen Speichers, die auf diese Art verloren geht, nennt man *interne Fragmentierung*. Die interne Fragmentierung wird durch die Wahl der Blockgrössen beeinflusst, in die die Freispeicherverwaltung die Halde aufteilt. Je

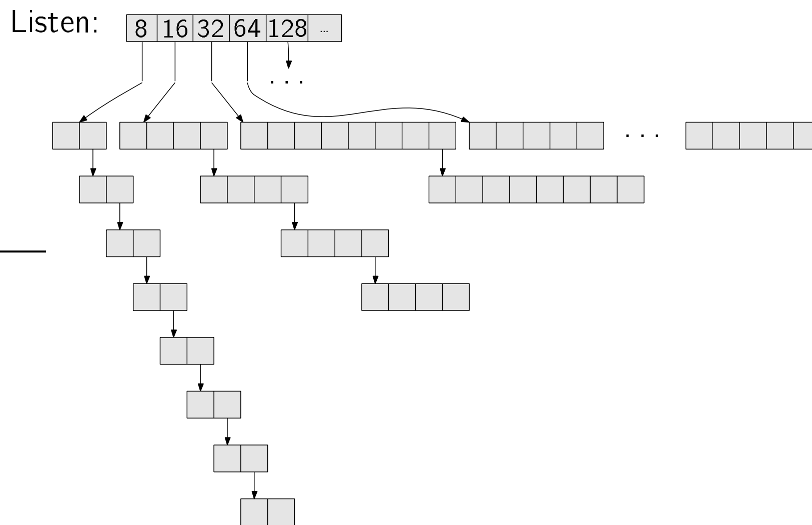


Abbildung 3.2: Freilisten

feiner diese Grössen gewählt werden, desto geringer werden regelmässig die Differenzen ausfallen; zum anderen bedeuten viele unterschiedliche Grössen auch, dass viele unterschiedliche Listen existieren. Dies bedeutet einen höheren Speicherbedarf für die vielen unterschiedlichen Listen. In [KP96] wird dieses Problem dadurch angegangen, dass für ein einzelnes Programm die Grössen der allozierten Speicherblöcke protokolliert werden; anhand dieses Protokolls wird eine Speicherverwaltung generiert, deren Blockgrössen auf das Programm angepasst sind.

Auffüllen der Listen Wenn eine Liste nach Durchführung einer Allokation leer ist, oder wenn bei der Allokation eine leere Liste angetroffen wird, muss diese Liste wieder aufgefüllt werden. Um die Liste wieder aufzufüllen, existieren folgende zwei Möglichkeiten: Entweder wird der Haldenspeicher vergrössert, oder die Liste wird mit grösseren Blöcken einer anderen Liste aufgefüllt.

Wird der Haldenspeicher vergrössert, so wird der neu hinzugekommene Bereich ganz in Blöcke der benötigten Grösse eingeteilt, mit denen die Liste aufgefüllt wird; dies birgt jedoch die Gefahr, dass der Haldenspeicher zügig wächst, auch wenn nur wenige Listen leerlaufen [WJNB95a].

Um die Liste mit Blöcken aus anderen Listen aufzufüllen, kommen natürlich nur die Listen in Frage, deren Blöcke eine Grösse haben, die ein ganzzahliges Vielfaches der benötigten Grösse sind. Wählt man zum Beispiel die Grössen als aufeinanderfolgende Zweierpotenzen [Kam98a, Lea96] (mit einer empirisch bestimmten Untergrenze), so kann man eine Liste immer mit Blöcken aus der Liste der nächstgrösseren Blöcke nachfüllen, indem man die grösseren Blöcke halbiert. Andere Techniken

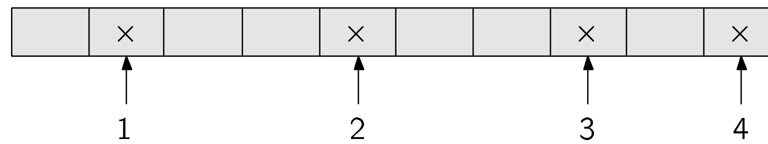


Abbildung 3.3: Beispiel für Externe Fragmentierung: Die Bereiche 1, 2, 3 und 4 sind belegt; insgesamt sind 6 Bereiche noch frei, jedoch kann kein Block angelegt werden, der grösser als 2 Bereiche ist.

wählen die Blockgrössen nach den Fibonacci-Zahlen [WJNB95a]. Läuft bei diesem Auffüllen die Liste der grösseren Blöcke leer, so kaskadiert das Auffüllen zur dann nächsten Liste. Setzt er sich bis zur letzten Liste fort, so muss der Haldenspeicher vergrössert werden, um die letzte Liste aufzufüllen.

Speicherrückgabe Gibt ein Programm einen Block an die Freispeicherverwaltung zurück, so muss er in die Liste eingefügt werden, in der Blöcke gleicher Grösse zusammengefasst sind. Um die Grösse eines Blockes erkennen zu können, muss diese innerhalb des Blockes gespeichert werden, jedoch an einer für das Programm nicht sichtbaren Stelle. Der Speicherplatz, der dadurch benötigt wird, wird manchmal separat als Verwaltungsaufwand (*overhead*) geführt, häufig jedoch zu der o. g. internen Fragmentierung hinzugezählt.

Externe Fragmentierung Nachdem einige Allokationen durchgeführt wurden, und nachdem einige der dabei angeforderten Speicherblöcke wieder an die Freispeicherverwaltung zurückgegeben wurden, stellt sich der lineare Haldenspeicher als eine im wesentlichen durch Zufall bestimmte Folge von Bereichen unterschiedlicher Längen dar, die entweder aus allozierten Blöcken oder aus freien Blöcken (die noch in einer der Freispeicherlisten eingeordnet sind) bestehen.

Betrachten wir die Bereiche, die aus freien Böcken bestehen, und beachten wir, in welchen Listen die Blöcke sind, so können wir den grössten Speicherblock finden, der in dieser Situation noch alloziert werden kann, ohne dass die Halde vergrössert werden muss. Ist dieser grösste Speicherblock nicht zufällig der einzige freie Block überhaupt, so ist die Summe der Grössen aller freien Blöcke grösser als die des grössten Speicherblockes.

Anfragen, die mehr Speicher als den des grössten Blockes, jedoch weniger als die Summe (der Grössen) der freien Blöcke fordern, können nicht ohne Vergrösserung der Halde beantwortet werden. Den Bereich, in dem sich solche Anfragen bewegen, nennt man *externe Fragmentierung*. In Abbildung 3.3 zeigen wir ein Beispiel.

Verschmelzen von Blöcken Nach wiederholtem Auffüllen von Freilisten kann es sein, dass die Freispeicherverwaltung mehr und mehr kleine Blöcke verwaltet. Werden diese Blöcke von dem Programm an die Freispeicherverwaltung zurückgegeben, so kann ihr Speicherplatz nicht dazu verwendet werden, Anfragen nach grösseren Blöcken zu beantworten.

Die Freispeicherverwaltung verwaltet dann eine grosse Menge an kleinen Blöcken, die möglicherweise nicht mehr vom Programm angefordert werden. Um Anfragen nach grösseren Blöcken zu beantworten, muss der Haldenspeicher vergrössert werden, um die Listen für Blöcke der geeigneten Grösse auffüllen zu können.

Um dies zu vermeiden, versucht man, die vorhandenen, kleinen Blöcke zu grösseren Blöcken zu verschmelzen. Dies wird durch Knuth's Randmarkierungen (engl. *boundary tags*, [Knu73]) möglich, die übrigens die im vorletzten Abschnitt erwähnte Grössenangabe innerhalb von Blöcken subsumieren.

3.4 Explizite Speicherverwaltung

Nachdem wir in Abschnitt 3.3 den grundlegenden Mechanismus zur Verwaltung des Haldenspeichers angegeben haben, widmen wir uns in diesem Abschnitt der Nutzung der Freispeicherverwaltung durch ein Programm.

Blöcke und Objekte Wir legen uns an dieser Stelle auf die Sprachregelung fest, dass das Programm in einem Speicherblock ein *Objekt* ablegt. Darunter fassen wir nicht nur die Objekte aus objektorientierten Sprachen zusammen, sondern auch die Verbundtypen aus nicht-objektorientierten Sprachen (wie `records` in *Pascal* oder `structs` in `C`). Aus Sicht des Programmes existieren nur Objekte, die in den Blöcken gespeichert werden, die von der Freispeicherverwaltung alloziert wurden. Da in jedem Block genau ein Objekt liegt, können wir nun Blöcke und Objekte miteinander identifizieren.

Allokation und Deallokation von Objekten Solange ein Objekt gebraucht wird, muss es alloziert bleiben; wird es nicht mehr gebraucht, muss es möglichst bald an die Freispeicherverwaltung zurückgegeben werden. Jedes Objekt hat also einen Lebenszyklus, wie er in Abbildung 3.4 dargestellt wird.

Während es üblicherweise sehr leicht ist, den Zeitpunkt zu bestimmen, an dem ein neues Objekt alloziert werden muss, ist es wesentlich schwieriger, den Zeitpunkt zur *De*allokation eines Objektes zu bestimmen, und dann auch noch die Deallokation geeignet umzusetzen.

Bei der expliziten Speicherverwaltung erarbeitet sich der Programmierer für jedes auf der Halde allozierte Objekt die Kenntnis darüber, wann der Zyklus aus

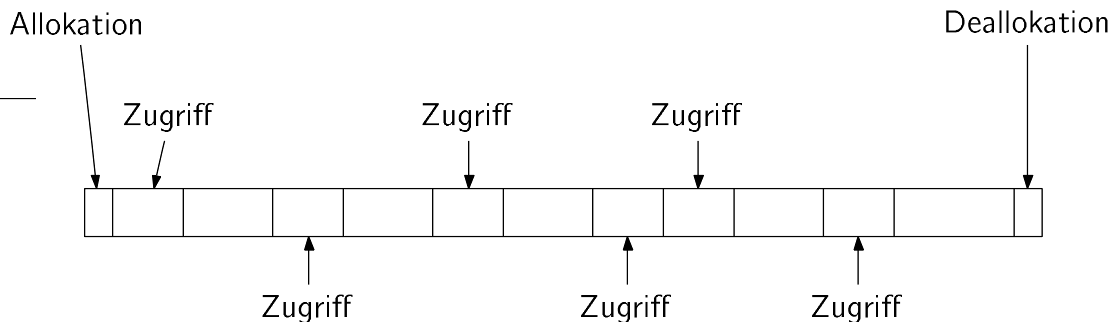


Abbildung 3.4: Lebenszyklus eines Objektes

Abbildung 3.4 durchlaufen ist und das Objekt gelöscht werden kann. An einem statischen Programmpunkt werden dann Anweisungen eingefügt, die das Objekt deallozieren.

Natürlich können zu diesem Zeitpunkt noch Referenzen auf das Objekt bestehen; diese dürfen natürlich nicht mehr dazu genutzt werden, um auf das Objekt zuzugreifen.

Zum anderen kann man bei Anweisungen, die auf ein Objekt zugreifen, häufig nicht präzise feststellen, dass dieser Zugriff der letzte Zugriff ist, der das Objekt erreicht. Ob noch weitere Zugriffe auf ein Objekt erfolgen, kann von dem noch bevorstehenden Ablauf des Programmes abhängen, und damit von Entscheidungen, deren Berechnung noch bevor steht.

Diese Objekte können erst zu einem Zeitpunkt dealloziert werden, zu dem alle möglichen Zugriffe *sicher* durchgeführt sind.

Oft ist es ratsam, die Existenz solcher Punkte durch geeignete Gestaltung der Struktur und Architektur eines Softwaresystems von vorneherein zu sichern.

Um die manuelle Speicherverwaltung eines Programmes zur Laufzeit zu untersuchen, existiert eine Vielzahl an Werkzeugen, die auf potentielle Fehler hinweisen können. Beispiele hierfür sind *ElectricFence*, *Mem-Test*, *Checker* [Piz99] und *Kefence* [JKSZ05]. Mit diesen Werkzeugen kann natürlich nicht eine korrekte Speicherverwaltung sichergestellt werden, aber bei der Suche nach Fehlern in der Speicherverwaltung können diese Werkzeuge nützlich sein.

Diese Vielfalt an Werkzeugen weist deutlich darauf hin, wie hoch der Bedarf an Hilfsmitteln ist, mit denen Fehler in der Speicherverwaltung gefunden werden können.

Objekt-Container Die Idee, Container-Datenstrukturen primär zum Zwecke der Speicherverwaltung einzusetzen, wird in [Han90] beschrieben. In [Kem95] erweitern

die Autoren durch die Verwendung von *variant records* in *Ada* 95 homogene Container zu heterogenen Containern, in denen semantisch verwandte Objekte zusammengefasst werden können. Arbeiten wie [TB97, GA98, CV98, BBD^{+}00, WSB01, BR01, GA01, QH02] stellen vergleichbare Ideen vor, die in verschiedener Art auf der Verwendung von *Regionen* basieren. Die Datenstruktur der „Region" wird uns in Kapitel 4 wieder begegnen, wo wir sie auch genauer beschreiben werden.

Mit der Verwendung von Containern zur Speicherverwaltung ergeben sich sowohl Vor- als auch Nachteile. Vorteilhaft ist, dass innerhalb der Software die Anweisungen, die Container erstellen und sie — einschließlich der darin enthaltenen Objekte — wieder verwerfen, das Programm strukturieren. Sie markieren Anfang und Ende von den Phasen, innerhalb derer die Objekte gebraucht werden, die in dem Container abgelegt werden. Daraus ergibt sich jedoch auch der Nachteil, dass durch die Platzierung dieser Anweisungen die Struktur der Software festgeschrieben wird, so dass sie bei späteren Umstrukturierungen [Bec99, Spo04] möglicherweise nur schwer zu ändern ist.

Ein weiterer Nachteil liegt darin, dass — mit der Ausnahme von *Reaps* [BZM02] — innerhalb von Containern nicht einzelne Objekte dealloziert werden können, auch wenn dem Programmierer die dazu nötigen Informationen zur Verfügung stehen.

Collections in Ada 83 Mit der Idee der *Collections* wurde in *Ada* 83 [Ada87] der Versuch unternommen, dem Übersetzer mehr Informationen über die mögliche Verwendung von Haldenobjekten zu geben, um ihm die Möglichkeit zu geben, die Lebenszeit einiger Haldenobjekte statisch zu bestimmen, und sie von der sonstigen Speicherverwaltung (Speicherbereiniger oder explizite Speicherverwaltung) auszunehmen. In *Ada* 95 [Ada95] existiert dieses Konzept zwar noch implizit, wurde aber mangels Interesse seitens der Ada-Gemeinschaft nicht weiter aufgeführt.

Spezialisierte Speicherverwaltungen Mit Überschneidung zum vorletzten Abschnitt nennen wir noch die Idee, die Speicherverwaltung zu spezialisieren, und sie in Teilen innerhalb des Programmes neu zu implementieren. In [Mye96] beschreibt der Autor die Nutzung des `new`-Operators für einen Allokator, der pro Klasse des Programms eine eigene Speicherverwaltung unterhält; sie nutzen dazu die *placement-new*-Erweiterung des `new`-Operators, die in [ES90] beschrieben ist. Ähnliche Vorgehensweisen finden sich auch in anderen Arbeiten [Cli93], allerdings primär mit dem Ziel, eine schnellere Speicherverwaltung zu erhalten, und nicht so sehr mit dem Ziel, den Programmierer bei der korrekten Bedienung der Speicherverwaltung zu unterstützen.

Interessanterweise berichten zuerst [WJNB95b] und später [BZM02], dass sich dadurch gerade gegenüber [Lea96, Kam98b], den wohl am weitesten verbreiteten

Implementierungen des Freilisten-Ansatzes, nur selten ein Laufzeitvorteil ergibt.

3.5 Automatische Speicherbereinigung

3.5.1 Funktionsweise

Das Ziel der automatischen Speicherbereinigung ist es, die Aufgabe der Speicherverwaltung vollständig automatisch durchzuführen, so dass sich der Programmierer damit nicht mehr befassen muss. Im Unterschied zur expliziten Speicherverwaltung wird hier der Speicher dynamisch, also zur Laufzeit reorganisiert, wobei allozierte Speicherblöcke, die nicht mehr benötigt werden, wieder der Freispeicherverwaltung unterstellt werden.

Freispeicherverwaltung für Speicherbereiniger Die meisten Speicherbereiniger können mit der Freispeicherverwaltung arbeiten, wie wir sie in Abschnitt 3.3 vorgestellt haben. Eine Ausnahme bilden Speicherbereiniger, die mit der „*Bereinigen durch Kopieren*“-Methode arbeiten, die wir später noch vorstellen. Während sich — abgesehen von der gerade genannten Ausnahme — die grundlegende Funktionsweise der Freispeicherverwaltung, die bei Speicherbereinigern eingesetzt wird, nicht von der unterscheidet, die wir in Abschnitt 3.3 vorgestellt haben, ist die Implementierung der Freispeicherverwaltung häufig mit der Implementierung des Speicherbereinigers abgestimmt. Insbesondere für *konservative* Speicherbereiniger, die wir ebenfalls später vorstellen, ergeben sich dadurch Laufzeit- und Effizienzvorteile.

Idee Dem Speicherbereinigern liegt folgende Beobachtung zu Grunde: Das Programm kann nur auf die Objekte zugreifen, die mit den Mitteln des Programmes erreichbar sind. Die Menge der erreichbaren Objekte ist damit eine Obermenge all derer Objekte, auf die das Programm im weiteren Ablauf noch zugreifen wird. Wir beschreiben in Folgenden, wie diese Menge berechnet wird.

Die Wurzelmenge Zunächst betrachtet man diejenigen Objekte, die für das Programm unmittelbar erreichbar sind. Referenzen zu diesen Objekten können sich im statischen Speicherbereich befinden, und in den Teilen der Aufrufschachteln, innerhalb derer die lokalen Variablen gespeichert sind. Die Menge der Referenzen, die innerhalb dieser beiden Bereiche angelegt sind, bezeichnet man hier als die *Wurzelmenge* (engl. *root set*).

Traversierung der Halde Objekte können Referenzen auf andere Objekte enthalten. Ausgehend von denjenigen Objekten, die man aus der Wurzelmenge heraus erreicht, folgt man rekursiv den Referenzen innerhalb der Objekte. Dabei traversiert man den Objektgraphen, den das Programm im bisherigen Verlauf angelegt hat. Die Traversierung beendet man dann, wenn man zwar alle Referenzen in allen gefundenen Objekten verfolgt hat, man dabei jedoch keine Referenz gefunden hat, die zu einem Objekt führt, das man noch nicht untersucht hat.

Bei der Traversierung des Objektgraphen führt man eine erschöpfende Suche über die Objekte aus. Der Zustand eines einzelnen Objektes in Bezug auf diese Suche wird in der Literatur [Wil90] gerne durch die Verwendung von drei Farben festgehalten. Den Objekten werden folgendermaßen Farben zugeteilt:

Definition 3.2 (Objektfarben während der Traversierung)

Weiss: *Ein weisses Objekt wurde noch nicht traversiert.*

Grau: *Ein graues Objekt wurde während der Traversierung gefunden, jedoch wurden noch nicht alle in diesem Objekt enthaltenen Referenzen verfolgt.*

Schwarz: *Ein schwarzes Objekt wurde gefunden, und alle von diesem Objekt ausgehenden Referenzen wurden verfolgt. Die damit erreichbaren Objekte wurden mindestens grau oder sogar schwarz gefärbt.*

Damit lässt sich das Grundprinzip der Traversierung folgendermaßen formulieren:

Definition 3.3 (Abstrakte Traversierung des Objektgraphen)

1. *Betrachte alle Objekte als weiss gefärbt.*
2. *Finde alle Objekte, die sich von Referenzen in der Wurzelmenge erreichen lassen, und färbe sie grau.*
3. *Solange es noch Objekte gibt, die grau gefärbt sind,*
   (a) *verfolge alle Referenzen eines grauen Objektes*
   (b) *sofern dabei Referenzen zu weissen Objekten gefunden werden, färbe diese Objekte grau.*
   (c) *nachdem alle Referenzen des grauen Objektes verfolgt wurden, färbe dieses Objekt schwarz.*
4. *Alle schwarz gefärbten Objekte sind für das Programm noch erreichbar, und können nicht dealloziert werden.*

5. *Alle noch weiss gefärbten Objekte sind für das Programm nicht mehr erreichbar, und können dealloziert werden.*

Die Objekte, die am Ende der Traversierung noch weiss gefärbt sind, bezeichnet man als Müll (engl. *garbage*).

Semantischer und syntaktischer Müll Mit der Idee, dass nur die Objekte alloziert bleiben müssen, auf die das Programm noch zugreifen *könnte*, hat man eine konservative Abschätzung geschaffen für diejenigen Objekte, auf die das Programm in seinem weiteren Ablauf *tatsächlich* noch zugreifen *wird*. Diejenigen Objekte, die zwar alloziert, aber nicht mehr erreichbar sind, bezeichnet man als *syntaktischen Müll*. Diese Objekte *zuzüglich* derjenigen Objekte, die zwar erreichbar sind, auf die das Programm jedoch nicht mehr zugreifen *wird*, bezeichnet man als *semantischen Müll*.

Sind grosse Datenstrukturen lange Zeit referenziert, obwohl sie für die Programmausführung nicht mehr von Bedeutung sind, zum Beispiel durch eine globale Variable, die im Verlauf des Programmes nie überschrieben wird, so können diese Datenstrukturen einen merklichen Unterschied zwischen dem syntaktischen Müll ausmachen, den der Speicherbereiniger beseitigen kann, und dem semantischen Müll, der durch einen Speicherbereiniger nicht erkannt werden kann.

Analysiert man den Speicherbedarf eines Programmes basierend auf Wissen über die Datenstrukturen, die während des Ablaufes angelegt werden, jedoch ohne in Betracht zu ziehen, welche Menge an Speicher durch semantischen Müll nicht dealloziert werden kann, so erhält man eine Unterschätzung des tatsächlichen Speicherbedarfs. Dies kann sich so auswirken, dass das Programm während des Laufes wegen Speichermangels abgebrochen wird, obwohl es auf einer Platform abläuft, die — laut ursprünglicher Schätzung — genug Speicher zur Verfügung stellt.

Die Drei-Farben-Invariante Während man die Traversierung durchführt, muss also ständig die Menge der aktuell grau gefärbten Objekte definiert sein. Wie dies in den konkreten Implementierungen geschieht, beschreiben wir weiter unten. Um die Korrektheit der Schlußfolgerungen Nr. 4 und Nr. 5 in Definition 3.3 sicherzustellen, muss während der Traversierung immer die im Folgenden angegebene *Drei-Farben-Invariante* erfüllt sein:

Definition 3.4 (Drei-Farben-Invariante) *Während der Traversierung in Definition 3.3 existieren nur Referenzen, die*

1. *von schwarz gefärbten Objekten auf schwarz oder grau gefärbte Objekte verweisen, oder*

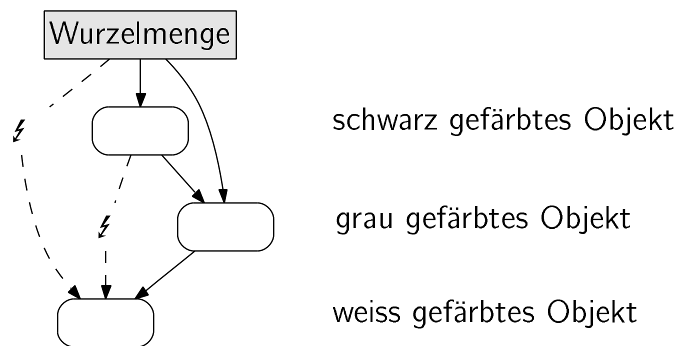


Abbildung 3.5: Drei-Farben-Invariante

2. *von grau gefärbten Objekten auf grau, schwarz oder weiss gefärbte Objekte verweisen.*

Es gibt also keine Referenzen, die von schwarzen Objekten aus auf weisse Objekte verweisen. Siehe auch Abbildung 3.5

Diese Invariante wird uns bei der Beschreibung von inkrementellen Speicherbereinigern hilfreich sein.

Speicherbedarf bei Speicherbereinigung Der Bedarf an Haldenspeicher, um ein Programm mit Speicherbereiniger ablaufen zu lassen, kann den Speicherbedarf des Programms deutlich übersteigen. Dieser Mehrbedarf kommt aus zwei Gründen zustande.

Zum Einen benötigt der Speicherbereinigung, selber Speicher, um die Reorganisation durchzuführen. Dieser Mehrbedarf kann zwischen 20%–50% betragen.

Zum Anderen steigt der Zeitaufwand vieler Implementierungen stark mit der Speicherauslastung *occ*:

Definition 3.5 (Speicherauslastung) *Die* Speicherauslastung *occ ist das Verhältnis zwischen verfügbarem Speicher* h_{total} *und alloziertem Speicher* h_{alloc}*:*

$$occ := h_{total}/h_{alloc}.$$

Je weniger freier Speicher zur Verfügung steht, desto öfter wird dieser durch das Programm ausgeschöpft, und desto öfter muss der Speicherbereiniger laufen. Da die Laufzeit des Speicherbereinigers proportional zum belegten Speicher ist, ergibt sich dann ein sehr schlechtes Verhältnis zwischen der Laufzeit des Speicherbereinigers und der Menge an Speicher, die er pro Lauf als frei melden kann.

Aus diesem Grund wird häufig bereits dann der Haldenspeicher durch Anforderung weiterer Speicherseiten vom Betriebssystem vergössert, wenn die Speicherauslastung eine vorbestimmte Marke überschreitet. Gängige Werte für diese Marke liegen im Bereich von $occ = 70\%$.

Präzise und konservative Speicherbereinigung In Definition 3.3 sind wir davon ausgegangen, dass sowohl die Wurzelmenge als auch der Objektgraph auf der Halde für den Speicherbereiniger klar zu erkennen sind. Dazu muss dem Speicherbereiniger bekannt sein, welche Variablen innerhalb der Wurzelmenge und auf der Halde tatsächlich Referenzen sind, und in welchen Variablen andere Werte (Fest- oder Gleitkomma-Zahlen, Zeichen und -ketten) gespeichert werden. Diese Informationen können an den Speicherbereiniger kommuniziert werden, indem bei der Übersetzung des Programmes zusätzliche Daten über die Belegung des statischen Speichers und über die Anordnung der lokalen Variablen in den Aufrufschachteln generiert werden; für die Objekte auf der Halde muss die Anordnung der Objektfelder bekannt sein. Bei polymorphen Sprachen muss zusätzlich noch der tatsächliche Typ eines Objekts erkennbar sein, wenn für den Typ der Referenz auf dieses Objekt im Programm mehrere konforme Typen existieren. Einen Speicherbereiniger, der diese Informationen zur Verfügung gestellt bekommt, und der sie vollständig nutzt, nennen wir einen *präzisen Speicherbereiniger*. Wie in [Bar88, BW88] gezeigt wird, ist es jedoch auch möglich, einen Speicherbereiniger einzusetzen, der nicht über diese Informationen verfügt. Mit Wissen über die Konventionen, mit der eine Hardwareplattform betrieben wird, und durch Verwendung von geeigneten Heuristiken ist es möglich, eine Traversierung zu schaffen, die *mindestens* den Objektgraphen umfasst, die aber aufgrund der Verwendung von Heuristiken potentiell mehr Speicherblöcke erreicht, als dies die Traversierung eines präzisen Speicherbereinigers tun würde. Wie in [Boe02] gezeigt wird, ist diese Differenz allerdings in vielen praktischen Fällen durchaus zu vertreten. Solche Speicherbereiniger nennen wir *konservative Speicherbereiniger*. Sie können in Umgebungen eingesetzt werden, in denen es schwierig oder unmöglich ist, die o. g. Informationen überhaupt zu erhalten (wie zum Beispiel bei der Verwendung von Programmiersprachen wie `C` und `C++`, die nicht typpräzise sind), oder in denen existierende Werkzeuge (Übersetzer, Binder, Lader) nicht dazu ausgerüstet sind, die benötigten Informationen zu liefern. Der Speicherbereiniger von [BW88] wird uns in Kapitel 5 noch einmal begegnen.

3.5.2 Umsetzung

Wie die in Def. 3.3 gegebene Traversierung des Objektgraphen umgesetzt wird, beschreiben wir in den folgenden Beispielen.

Markieren und Abräumen Die wohl direkteste Anwendung von Def. 3.3 zur Speicherbereinigung ist eine Strategie, die als *Markieren und Abräumen* (engl. *mark and sweep*) bekannt ist [Wil90, WJ93, Joh97]. Nach Durchführung der Traversierung werden alle Objekte, die noch weiss gefärbt sind, die jedoch von der Freispeicherverwaltung als alloziert geführt werden, wieder an die Freispeicherverwaltung zurückgegeben. Diese Strategie kommt meistens auch in konservativen Speicherbereinigern wie in [BW88] zum Einsatz. Durch geschickte Organisation der Freispeicherverwaltung kann man erreichen, dass die Laufzeit für eine Traversierung des Objektgraphen nicht von der Grösse des Haldenspeichers abhängt (alle schwarzen Objekte werden bei der Traversierung inspiziert, und alle weissen Objekte beim Abräumen), sondern nur von der Grösse des Objektgraphen (nach Inspektion der schwarzen Objekte sind die weissen Objekte bereits als frei an die Freispeicherverwaltung gemeldet).

Markieren und Kompaktieren Eine Abwandlung der *mark-and-sweep*-Strategie ist die *Markieren und Kompaktieren*-Strategie. Der Unterschied besteht darin, dass jedes Objekt im Haldenspeicher verschoben wird, wenn es grau gefärbt wird. Das Ziel dieser Verschiebung wird so gewählt, dass innerhalb der Freispeicherverwaltung die Gefahr der externen Fragmentierung gemindert wird. Alle Referenzen auf dieses Objekt, die der Speicherbereiniger bei der Traversierung findet, müssen so umgeschrieben werden, dass sie auf die erstellte Kopie des Objektes zeigen. So wie die vorige Strategie kann diese Strategie mit einer Freispeicherverwaltung mit Freilisten eingesetzt werden. Diese Strategie kann bei konservativen Speicherbereinigern nicht eingesetzt werden, da diese nicht nur die Referenzen identifizieren, die im Objektgraph tatsächlich vorhanden sind, sondern möglicherweise auch Werte als Referenzen behandeln, die keine sind. Wenn Speicherblöcke als erreichbar erkannt werden (also durch den Speicherbereiniger grau und schwarz gefärbt werden), so führt dies nicht zu Fehlern, sondern nur zu einer geringeren Effektivität des Speicherbereinigers; würde der Speicherbereiniger jedoch Werte umschreiben, die keine Zeiger sind, so würde dies die Korrektheit beeinträchtigen.

Bereinigen durch Kopieren Diese Strategie hat mit der vorigen Strategie gemeinsam, dass grau gefärbte Objekte kopiert werden. Allerdings setzt sie nicht auf eine Freispeicherverwaltung mit Freilisten auf, sondern auf eine eigene Speicherverwaltung: Der Haldenspeicher wird in zwei gleich grosse Teile aufgeteilt. Den einen Teil nennt man *from-space*, den anderen *to-space*. Die Allokation von Objekten findet nur im *from-space* statt; während der Speicherbereinigung werden Objekte vom *from-space* in den *to-space* kopiert. Nach einem kompletten Durchlauf befinden sich alle schwarzen Objekte im *to-space*. Die beiden Teile vertauschen dann die

Rollen, und das Programm kann im ehemaligen *to-space*, dem jetzt neuen *from-space*, Objekte allozieren.

Innerhalb des jeweiligen *from-space* lässt sich eine besonders einfache Speicherverwaltung realisieren: Die Speicherverwaltung unterhält ständig einen Zeiger in den *from-space*. Dieser Zeiger zeigt anfangs auf die erste Addresse des *from-space*; für die Allokation von n Bytes wird sein aktueller Wert zurückgegeben, und der Zeiger wird um n Bytes erhöht. Bei dieser Art der Speicherverwaltung kann weder externe noch interne[1] Fragmentierung auftreten.

Ein Nachteil dieser Methode besteht darin, dass — zumindest während der Bereinigung — die zur Verfügung stehende Halde mindestens doppelt so gross sein muss, wie der Speicher, den das Programm alloziert hat, und dass der Kopiervorgang potentiell eine sehr geringe Lokalität hat, so dass die Gefahr steigt, dass während des Kopiervorganges viele Seitenwechsel auftreten. Mit dem *Semi-Space Collector* umgeht Bartlett [Bar90] dieses Problem, indem er die beiden Teile des Speichers nicht als lineare Speicherbereiche organisiert, sondern in einzelne Segmente aufteilt, die einzeln und damit mit einer höheren Lokalität in den Hauptspeicher eingelagert werden können.

3.5.3 Inkrementelle Speicherbereinigung

Bei der Anwendung von Def. 3.3, wie dies auch in den in Abschnitt 3.5.2 beschriebenen Verfahren geschieht, ergibt sich für die Traversierung ein Zeitaufwand, der — mindestens — proportional zu der Anzahl der Objekte ist, die den Objektgraphen auf der Halde ausmachen. Dadurch kann es während der Ausführung des Programmes bei der Durchführung der Traversierung zu einer merklichen Wartezeit kommen. Bei manchen Einsatzzwecken von Software kann dies zu Problemen führen: Bei Software, mit der Menschen direkt interagieren (zum Beispiel graphische Nutzungsoberflächen), könnte der Nutzer durch das scheinbare „Einfrieren" des Programms den Eindruck bekommen, das Programm sei „abgestürzt" und das System reagiere nicht mehr; bei Software, die einen kontinuierlichen Durchsatz bei der Bearbeitung ihrer Eingabe aufrechterhalten soll, insbesondere die weichen Echtzeitsysteme, wie wir sie in Abschnitt 3.1.1 beschrieben haben, kann während der Traversierung der gewünschte Durchsatz nicht aufrecht erhalten werden. Bei harten Echtzeitsystemen kann eine Traversierung so lange dauern, dass eine Zeitschranke nicht eingehalten werden kann.

In diesen Fällen kann man Speicherbereiniger einsetzen, die *inkrementell* arbeiten [Bak78, LH83, Wil90, Bak92, Nil93, WJ93, Joh97, Sie98, Sie99, CB01,

[1] Speicherverschnitt, der deshalb auftritt, weil die zu Grunde liegende Hardware einen Mindestwert für die Ausrichtung von Objekten im Speicher fordert, zählen wir hier nicht als Fragmentierung.

BCR03, Rit03]. Die Arbeit, die für die Traversierung anfällt, wird hierbei in Teilaufgaben aufgeteilt, wobei die Dauer zur Abarbeitung einer einzelnen Teilaufgabe — mit Blick auf den Einsatzzweck — kurz genug ist. Zwischen der Abarbeitung der Teilaufgaben kann das Programm weiter laufen.

Einhaltung der Drei-Farben-Invariante Wenn die Arbeit des Speicherbereinigers durch das Programm unterbrochen wird, kann die Drei-Farben-Invariante verletzt werden. Das Programm kann in ein bereits schwarz gefärbtes Objekt einen Zeiger auf ein weisses Objekt schreiben. Das weisse Objekt ist dann für das Programm erreichbar; ist jedoch die Referenz aus dem schwarzen Objekt die einzige Referenz auf das weisse Objekt, so wird — da der Speicherbereiniger das schwarze Objekt nicht noch einmal inspizieren wird — das weisse Objekt nicht als erreichbar erkannt und fehlerhafterweise freigegeben.

Um sicher zu stellen, dass die Drei-Farben-Invariante eingehalten wird, müssen solche Speicherzugriffe des Programmes erkannt und geeignet behandelt werden.

Erkennen von Verletzungen der Drei-Farben-Invariante Speicherzugriffe, die die Drei-Farben-Invariante verletzen, kann man mit Schreib- oder Lesesperren erkennen. Diese Sperren bewachen Zugriffe auf Objekte, und lösen beim Zugriff auf ein Objekt Aktionen des Speicherbereinigers aus. Schreibsperren [Wil90, WJ93] bewachen schreibende Zugriffe und werden auf schwarz und grau gefärbte Objekte gesetzt, Lesesperren [Bak78, Bak92] bewachen lesende Zugriffe und werden auf weisse Objekte gesetzt.

Behandlung von Verletzungen der Drei-Farben-Invariante Die Aktion, die im Speicherbereiniger ausgelöst wird, inspiziert die Addressen, die in dem Lese- bzw. in dem Schreibzugriff verwendet wurden; handelt es sich dann um einen Zugriff, der die Drei-Farben-Invariante verletzt oder der sie nur verletzen könnte (ein schreibender Zugriff auf ein schwarzes oder graues Objekt oder das Lesen einer Referenz auf ein weisses Objekt), so müssen die dabei betroffenen Objekte geeignet behandelt werden. Es gibt dazu zwei unterschiedliche Strategien:

Bestandsaufnahme am Anfang: Hier wird folgende, zusätzliche Invariante aufrecht erhalten: Alle Objekte, die zu Beginn einer Traversierung erreichbar waren, bleiben während der Traversierung erreichbar. Um dies sicher zu stellen, wird jedes Mal, wenn im Objektgraphen ein Pfad $a \rightarrow b$ gebrochen wird, ein neuer Pfad zu Objekt b künstlich hergestellt. Möglicherweise werden Objekte — durch die künstlich geschaffenen Pfade — als erreichbar angesehen, obwohl sie über den ursprünglichen Objektgraph nicht mehr erreichbar sind.

Schrittweise Aktualisierung: Wird eine Referenz $a \to b$ erstellt, wobei a schwarz gefärbt ist, muss entweder a oder b wieder grau gefärbt werden, so dass es erneut vollständig vom Speicherbereiniger inspiziert wird. Die erneut grau gefärbten Objekte können entweder in einer separaten Arbeitsliste eingereiht werden, die der Speicherbereiniger abarbeitet, oder sie können dem Speicherbereiniger direkt in den Prozess der Traversierung eingeschleust werden.

Einteilung der Arbeitszeit Den Umfang der einzelnen Teilaufgaben — Abarbeiten der Wurzelmenge, Objekte markieren, Objekte kopieren oder die als frei erkannten Speicherblöcke der Freispeicherverwaltung übergeben — kann man statisch oder dynamisch festlegen. Ein Beispiel für einen statischen Umfang wäre eine maximale Anzahl von Objekten, die inspiziert werden, bevor der Speicherbereiniger die Arbeit unterbricht und das Programm weiterlaufen kann [Bak78]. Die dynamische Aufteilung kann auf Grössen zurückgreifen, die erst bei der Ausführung des Programmes zur Verfügung stehen, wie zum Beispiel der Füllgrad des Speichers, oder das kurz- oder mittelfristig zurückliegende Verhalten des Programmes in Bezug auf die Allokationsrate (Anzahl der Objekte pro Zeiteinheit, oder Summe der Grössen der allozierten Objekte). Oft wird das Verhalten des Programmes über den Zeitraum in Betracht gezogen, der mit dem Abschluß der letzten vollständigen Traversierung begonnen hat [Sie98, Sie99]. Eine andere Möglichkeit, den Umfang der Teilaufgaben des Speicherbereinigers zu bestimmen, besteht darin, für den Speicherbereiniger einen eigenen Faden zu unterhalten. Die Priorität, die diesem Faden zugewiesen wird, bestimmt dann indirekt, welcher Anteil an der gesamten Laufzeit dem Speicherbereiniger-Faden zugeteilt wird [WJ93, Boe00, CB01, BCR03]. Während der Laufzeit kann man die Priorität des Fadens dann variieren, zum Beispiel mit dem Füllgrad des Freispeichers, oder der aktuellen Allokationsrate des Programms.

Mit inkrementellen Speicherbereinigern kann man die anfallende Arbeit zu einem gewissen Maße über die Laufzeit des Programms verteilen. Die Arbeiten von [Sie98, Sie99] zeigen zum Beispiel sehr deutlich, dass der Speicherbereiniger mit nur einem sehr geringen Anteil an der gesamten Laufzeit des Systemes auskommt, sofern der Belegungsgrad der Halde ein gewisses Maß nicht überschreitet. Für einen Belegungsgrad von 70% zum Beispiel muss der Speicherbereiniger von [Sie99] nur durchschnittlich 15.72 Blöcke à 32 Byte pro alloziertem Block inspizieren.

Freigabe des Speichers nach Ende der Traversierung Ein inkrementeller Speicherbereiniger führt zwar seine Arbeit in einzelnen Schritten durch, jedoch liefert er sein Ergebnis nicht schrittweise, sondern erst nach Ende einer vollständigen Traversierung des Objektgraphen. Erst nachdem der Objektgraph vollständig traversiert wurde, sind *sicher* alle erreichbaren Objekte schwarz eingefärbt, und erst

dann können alle noch weiss eingefärbten Objekt als frei betrachtet werden. Während also der Speicherbereiniger — verzahnt mit dem Programm — voranschreitet, kann das Programm kontinuierlich neuen Speicher allozieren, während mehr und mehr früher allozierte Objekte nicht mehr erreichbar werden. Diese Objekte können aber erst dann wieder in die Freispeicherverwaltung eingegliedert werden, wenn der Speicherbereiniger seine Traversierung beendet hat.

Das regelmässige Voranschreiten der Arbeit des Speicherbereinigers liefert also nicht etwa mit gleicher Regelmässigkeit neu entdeckten freien Speicher, sondern immer nur dann, wenn eine Traversierung vollständig durchgeführt wurde. Das sieht man wiederum sehr deutlich bei [Sie99]: Bei einem Belegungsgrad des Hauptspeichers von 90% muss der Speicherbereiniger bereits durchschnittlich 64.21 Blöcke für jeden allozierten Block inspizieren, bei 95% sind es dabei schon 137.9 Blöcke; mit dieser Steigerung erreicht [Sie99], dass — gegen eine bekannte Allokationsrate des Programmes — die Traversierung des Speicherbereinigers rechtzeitig beendet ist, bevor das Programm den verbleibenden Speicher durch Allokationen auch noch belegt hat, so dass eine Allokation aus Mangel an freiem Speicher nicht beantwortet werden kann.

Speicherbereiniger in Echtzeitsystemen Für die Speicherverwaltung von Echtzeitsystemen interessieren wir uns, wie bereits in Abschnitt 3.1.2 betont, nicht für die durchschnittliche, sondern immer nur für die maximale Laufzeit. Es gibt in der Literatur Ansätze, die eine Garantie geben dafür, dass immer genug Freispeicher zur Verfügung steht, wenn zum Beispiel dem Speicherbereiniger nur genug Arbeitszeit eingeräumt wird, auch wenn manchmal wiederum nicht ganz klar ist, ob die Speicherbereiniger nun für „weiche" oder für „harte" Echtzeitsysteme geeignet sind [Det04].

Problematisch ist bei diesen Ansätzen jedoch, dass sie eine sehr geringe Effizienz haben. Genauer gesagt ist die beste Abschätzung ihrer maximalen Laufzeit (im Sinne der *WCET* bei Echtzeitsystemen) durchgehend weitaus höher als die durchschnittliche Laufzeit, die man regelmässig in Systemen beobachten, die Speicherbereiniger verwenden.

In [BCR03], der neuesten Arbeit, die uns auf diesem Gebiet bekannt ist, heben die Autoren besonders auf die hohe Laufzeiteffizienz ab. Die existierenden Arbeiten haben die Autoren von [BCR03] folgendermassen zusammengefasst:

> We [...] show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application, a factor of 4 improvement in utilization over the best previously published results.

Mit anderen Worten, dem Speicherbereiniger muss mehr Rechenzeit eingeräumt werden als dem Programm, und der zusätzliche Speicherbedarf für das gesamte System liegt wesentlich über dem eigentlichen Speicherverbrauch des Programmes.

Zusammenfassung Automatische Speicherbereiniger sind eine sinnvolle Alternative zur manuellen Speicherverwaltung. Sie ersparen dem Programmierer viel Aufwand bei der Erstellung und insbesondere bei der Weiterentwicklung von Software. Ihre Laufzeit ist zu der der manuellen Speicherverwaltung durchaus konkurrenzfähig, wenn auch nicht immer transparent. Speicherbereiniger lassen sich auch für Echtzeitsysteme einsetzen. In diesem Bereich bieten sie zwar durch ihre Konstruktion den Schutz vor Speicherfehlern, der in sicherheitskritischen Systemen unerlässlich ist; die Laufzeit jedoch, die man dem Speicherbereiniger zuteilen muss, damit die Einhaltung von Zeitschranken nicht gefährdet ist, ist jedoch — im Gegensatz zu der tatsächlich benötigten Laufzeit — sehr hoch, und die Auslastung des Systems damit sehr niedrig.

3.6 Automatische Speicherverwaltung in funktionalen Programmen

Für die Speicherverwaltung von funktionalen Programmen existiert eine Vielzahl von Ansätzen [TT93, TT94, TB97, TB98, GA98, Hal99, NKY04].

Ausführung von funktionalen Programmen Bei der Ausführung von funktionalen Programmen wird der aktuelle Zustand der Ausführung durch Objekte repräsentiert. Diese Objekte stehen für Instanziierungen (Aufrufe) von Funktionen, und für die Anwendung von Operationen und Typkonstruktoren. Diese Elemente, die während der Ausführung erstellt werden, können sich bei ihrer Erstellung nur auf bereits existierende Objekte beziehen, und sie können während der weiteren Ausführung des Programms nicht destruktiv geändert werden. Damit haben alle Objekte, die während der Ausführung eines funktionalen Programmes angelegt werden, die Eigenschaft, dass alle Referenzen, die in diesen Objekten enthalten sind, immer auf ältere Objekte zeigen.

Escape-Analyse Die Eigenschaft, dass hier Objekte nur ältere Objekte referenzieren können, kann man sich bei der Speicherverwaltung zu Nutze machen. Durch Analyse der Allokationszeitpunkte der erstellten Objekte, und durch Analyse der Auswertungsreihenfolge kann man eine konservative Abschätzung erhalten, wie lange die einzelnen Objekte alloziert bleiben müssen. Diese Analyse wird, genau wie ihr

Pendant im Bereich der imperativen Sprachen [Bla99, GS00a, GS00b] als *Escape-Analyse* bezeichnet.

Umsetzung Die Lebensdauer der Objekte wird eingefangen durch die Lebensdauer einer geeigneten Funktionsinstanziierung oder einer Termauswertung. Die Objekte werden — wie wir dies bereits für imperative Sprachen am Ende von Abschnitt 3.4 beschrieben haben — in Containern gesammelt; diese Container werden erstellt am Anfang der Funktion bzw. der Termauswertung, die die Lebenszeit der Objekte umfaßt, und sie werden am Ende der Funktions- bzw. Termauswertung wieder verworfen. Manche Ansätze, wie [Hal99], verlassen sich zusätzlich noch auf einen Speicherbereiniger, während andere Ansätze, wie [TT93, TT94, TB97, TB98, GA98, NKY04] sich ausschließlich auf die Ergebnisse der Escape-Analyse verlassen.

Verwendung für imperative Sprachen Bei funktionalen Sprachen können also mittels der Escape-Analyse Lebenszeiten für Objekte berechnet werden, und basierend auf diesen Lebenszeiten kann eine statische Speicherverwaltung vollständig automatisch errechnet werden. Es stellt sich die Frage, ob sich die hier so erfolgreichen Ansätze auch für imperative Sprachen einsetzen lassen.

Diese Ansätze liefern ihre Informationen für Programme, die in einer funktionalen Sprache geschrieben sind, während wir uns um eine Methode bemühen, die sich mit imperativen Sprachen beschäftigt.

Um diese Ansätze auf ein imperatives Programm P_{imp} zuwenden, müsste ein äquivalentes funktionales Programm P_{func} geschaffen werden. Obwohl es immer möglich ist, einen gegebenen *Algorithmus* $\mathcal{A}$ sowohl funktional als $P_{func}(\mathcal{A})$ und imperativ als $P_{imp}(\mathcal{A})$ zu implementieren, ist es nicht möglich, zu einem gegebenen $P_{imp}(\mathcal{A})$ den darin implementierten Algorithmus und dafür ein äquivalentes $P_{func}(\mathcal{A})$ zu bestimmen. Spätestens die Analyse der Instruktionen, die mit Referenzen arbeiten, wird, wie in [Ram94] gezeigt, scheitern.

Ansätze wie [JW93, Jon00], erlauben zwar die Verwendung von imperativen Elemente in funktionalen Sprachen, und ermöglichen damit die kompositionale Übersetzung von imperativen in funktionale Programme; damit verlieren jedoch die Eingangs erwähnten Ansätze an Wert, da es innerhalb von funktionalen Sprachen kein Gegenstück zu der Datenstruktur gibt, die in imperativen Sprachen als Haldenspeicher zur Verfügung steht. Innerhalb der funktionalen Sprache muss er jedoch durch eine eigene Datenstruktur implementiert werden. Dadurch übertragen sich jedoch wiederum alle Probleme mit der Verwaltung des Speichers auf das funktionale Programm, die in der imperativen Implementierung bereits existierten.

Original:	**Mit Referenzzähler:**
`p = q`	see code below

```
if (NULL != p) {
   p -> refcount --;
   if (0 == p -> refcount) {
     rc_free (p);
   }
}

if (NULL != q) {
   q -> refcount ++;
}

p = q;
```

Abbildung 3.6: Aktualisierungen von Referenzzählern bei einer Zuweisung. Die Funktion `rc_free ()` übernimmt die rekursive Löschung des Objektes.

3.7 Referenzzähler

Diese Methode wurde zuerst von G. E. Collins in [Col60] vorgestellt. Die Beobachtung, die der Technik der Referenzzähler zu Grunde liegt, ist folgende: Wenn es keine Referenzen auf ein Objekt mehr gibt, so ist dieses Objekt nicht mehr erreichbar und kann dealloziert werden. Dies wird dadurch verfolgt, dass für jedes Objekt ein Zähler unterhalten wird. Dieser Zähler wird bei der Allokation des Objektes auf 0 gesetzt. Wird eine neue Referenz auf das Objekt gesetzt, so wird er um 1 erhöht; verschwindet eine Referenz, so wird er um 1 erniedrigt. Fällt der Referenzzähler dabei auf 0, so wird das Objekt dealloziert.

Umsetzung Da der Referenzzähler immer dann geändert werden muss, wenn Referenzen auf ein Objekt gesetzt werden oder wenn Referenzen verschwinden, müssen bei allen Anweisungen des Programms, die dies verursachen könnten, die nötigen Aktualisierungen der betroffenen Referenzzähler eingefügt werden. Zu beachten sind hier alle Anweisungen, die referenzwertige Variablen beschreiben. In Abbildung 3.6 zeigen wir dies am Beispiel einer Zuweisung zwischen zwei Referenzen[2].

Verzögerte Referenzzähler Wie man in Abbildung 3.6 erkennen kann, wird durch die Anweisungen, die Referenzzähler aktualisieren, aus einer einfachen Zuweisung

[2] Dieses Beispiel haben wir einem Vortrag von H.P. Boehm [BW88, Boe00, Boe02] entnommen.

eine deutlich umfangreichere Berechnung. Werden die Referenzzähler durch eine Implementierungstechnik auf Quellsprachebene eingesetzt (wie zum Beispiel durch sog. *smart pointers* in `C++` [Mye96, Mye97]), so kann dies den Laufzeitbedarf stark in die Höhe treiben. Wie man aus Abbildung 3.6 entnimmt, entstehen bei einer einfachen Zuweisung zwischen zwei lokalen Variablen bereits — unter anderem — vier Speicherzugriffe auf zwei Objekte, von denen potentiell zwei Zugriffe nicht durch den Cache abgefangen werden können.

Werden diese Anweisungen jedoch automatisch eingesetzt, zum Beispiel durch den Übersetzer [Rit03], so lassen sich häufig einige Aktualisierungen gegeneinander aufheben. Wir illustrieren dies mit Beispiel 3.6.

Beispiel 3.6 (Verzögerte Referenzzähler) *Gegeben sind die Variablen `p` und `q`. Wir möchten ihren Inhalt vertauschen. Der Referenzzähler eines Objektes sei in dem Feld „`rc`“ gespeichert. Wir verwenden die temporäre Variable `t` und „tauschen im Dreieck“:*

Code	Referenzzähler			Kommentar
{				
Var `t;`				
`t = p`	`p->rc ++`			
`p = q`	`p->rc --`	`q->rc ++`		
`q = t`		`q->rc --`	`t->rc ++`	
}			`t->rc --`	*Variable `t` verlässt den Sichtbarkeitsbereich*

Die Aktualisierungen der Referenzzähler der beiden Objekte haben wir neben dem eigentlichen Programmcode aufgeführt. Wie man sieht, werden zwar insgesamt sechs Mal Referenzzähler aktualisiert, jedoch heben sich die Aktualisierungen jeweils paarweise wieder weg.

Kaskadierende Deallokation Wird ein Objekt gelöscht, so verschwinden damit auch die Referenzen, die innerhalb dieses Objektes gespeichert waren. Die dadurch möglicherweise referenzieren Objekte verlieren also eine Referenz, und ihre Referenzzähler müssen erniedrigt werden. In einigen Fällen kann dies dazu führen, dass dabei Referenzzähler auf 0 fallen, und weitere Objekte dealloziert werden müssen. Die Deallokationen können kaskadieren und das Programm an dieser Stelle (scheinbar) zu einer Pause veranlassen. In Abbildung 3.7 ist ein Beispiel gezeigt.

Da in Echtzeitsystem der Umfang von Datenstrukturen bekannt sein muss, ist auch die Dauer dieser Unterbrechung vorhersagbar. Nicht ohne Weiteres vorhersagbar ist jedoch, *wann* diese Unterbrechung auftritt. Um den Zeitpunkt zu bestimmen, muss bekannt sein, wann die letzte Referenz auf die Datenstruktur verschwindet.

Beispiel 3.7 *Die letzte Referenz auf die Wurzel eines binären Baumes verschwindet, und es gibt keine weiteren Referenzen auf andere Knoten des Baumes. Zunächst fällt der Referenzzähler der Wurzel auf* 0*, und der Wurzelknoten wird dealloziert. Damit fallen auch die Referenzzähler der Kinder der Wurzel auf* 0*. Mit deren Deallokation wiederum fallen die Referenzzähler deren Kinder auf* 0*. Diese Reaktion setzt sich fort, bis schließlich die Blätter des Baumes dealloziert werden.*

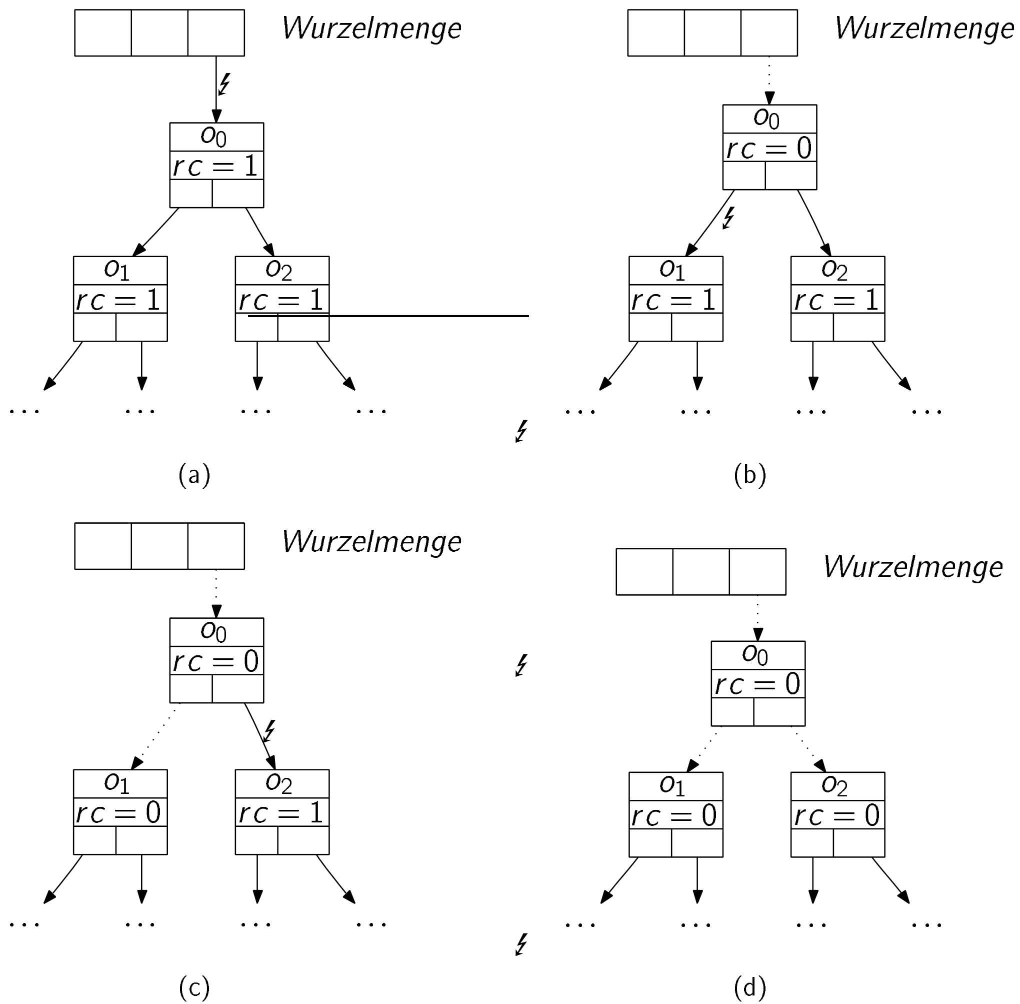


Abbildung 3.7: Beispiel für kaskadierende Löschung bei Referenzzählern

Wenn dies jedoch immer bekannt ist, kann man genauso gut an eben dieser Stelle die Datenstruktur explizit löschen. Damit können aber Referenzzähler keinen Vorteil mehr bieten.

Faules Deallozieren Den Effekt der kaskadierenden Deallokation kann man dadurch abmildern, dass man die Referenzen eines deallozierten Objektes nicht sofort abarbeitet; vielmehr wird das Objekt bei der Deallokation zunächst auf eine gesonderte Warteliste gesetzt. Objekte auf dieser Warteliste kann man schrittweise abarbeiten, zum Beispiel wenn die eigentliche Freiliste leer ist, oder in einem Zeitraum, bei dem innerhalb des Programmes bekannt ist, dass momentan keine Zeitschranken einzuhalten sind, oder während das Programm aufgrund von Auftragsmangel im Leerlauf ist.

Der Aufwand für das Freigeben eines Objekts auf der Warteliste ist dann begrenzt durch die maximale Anzahl von Referenzen, die in diesem Objekt existieren kann. In Systemen, in denen alle Objekte die gleiche Grösse haben, wie zum Beispiel in Interpretern für die Programmiersprache *Lisp*, kann immer durch das Freigeben eines einzigen Objektes von der Warteschlange eine anstehende Allokation beantwortet werden, wobei in jedem Fall eine Referenz berücksichtigt werden muss. Wenn die Objekte, die in der Warteliste eingetragen sind, unterschiedliche Grössen haben können, müssen dann allerdings immer noch solange Objekte dealloziert werden, bis ein Objekt dealloziert wurde, mit dem genug Speicherplatz freigegeben wird, um die anstehende Allokation zu beantworten.

Zyklen im Objektgraphen Ein entscheidender Nachteil von Referenzzählern ist, dass sie Objekte immer dann nicht deallozieren können, wenn diese Objekte Teil eines Zyklus' innerhalb des Objetkgraphen sind. Dieses Problem wurde zuerst von J. H. McBeth in [McB64] beschrieben.

Verschwindet die letzte Referenz, die „von aussen" in einen Zyklus hineinweist, so bestehen immernoch die Referenzen der Objekte innerhalb des Zyklus' untereinander, und die Referenzzähler dieser Objekte fallen nie auf 0, so dass die Objekte nie dealloziert werden.

Für dieses Problem findet sich in der Literatur keine Lösung, die *innerhalb* der Referenzzähler angesiedelt ist. Es existieren jedoch folgende Ansätze:

Zusätzliche Verwendung eines Speicherbereinigers: Zusätzlich zur Technik des Referenzenzählens wird ein Speicherbereiniger eingesetzt, der diejenigen Objekte identifiziert, die als Bestandteil von Zyklen im Objektgraph von dem Referenzzähler verloren wurden, und wieder der Freispeicherverwaltung übergibt.

Automatische Erkennung von Zyklen: Durch Analysen wie zum Beispiel die in [HHDH02, Hir04] kann man diejenigen Anweisungen des Programmes identifizieren, die möglicherweise Zyklen erstellen. Es existieren unseres Wissens noch keine Ansätze, die über diese Idee hinausgehen.

Einschränkungen der Ausdrucksmächtigeit: Die pragmatische Herangehensweise besteht darin, dass man verbietet, Programme zu schreiben, die Zyklen im Objektgraphen erstellen.

Überlauf der Referenzzähler Wählt man zur Implementierung der Referenzzähler einen ganzzahligen Datentyp, der (mindestens) so breit wie eine Speicher-Adresse ist, so kann ein Referenzzähler nicht überlaufen. Ist die Breite der Referenzzähler geringer, so besteht die Möglichkeit, dass der Referenzzähler eines Objektes überläuft. Wenn ein Referenzzähler überläuft, ist es natürlich nicht mehr sinnvoll, ihn zu dekrementieren, wenn Referenzen auf das Objekt verloren gehen. Stattdessen bleibt der Zähler auf seinem Wert, und man verlässt sich zur Deallokation der Objekte, deren Referenzzähler übergelaufen sind, auf einen Speicherbereiniger. In [RW77, RW98] beschreiben die Autoren die Beobachtung, dass der Wert, den ein Referenzzähler am häufigsten annimmt, der Wert 1 ist, dass also die meisten Objekte von genau einer einzigen Referenz aus erreichbar sind. Als Konsequenz ergibt sich, dass selbst mit einem Referenzzähler, der nur ein einziges Bit umfasst, ein grosser Teil der Objekte dann dealloziert werden kann, wenn die (einzige und) letzte Referenz verloren geht. Das Bit, das hierfür benötigt wird, lässt sich häufig innerhalb des Datenbereichs integrieren, der für ein Objekt angelegt werden muss, so dass sich der Speicherbedarf nicht erhöht.

Zusammenfassung Zum Teil wegen des oft unerwartet hohen Rechenaufwandes, hauptsächlich jedoch aufgrund der Unfähigkeit von Referenzzählern, mit Zyklen im Objektgraphen umzugehen, sind Referenzzähler nicht als einziges Mittel der automatischen Speicherverwaltung zu gebrauchen. Als Teil einer umfangreicheren Infrastruktur wie die von [Rit03] jedoch, oder als automatische Speicherverwaltung für eine sinnvoll begrenzte Teilmenge der Objekte (zum Beispiel für alle Objekte, die selber keine Referenzen enthalten), können Referenzzähler gute Dienste leisten.

3.8 Übersetzer-Optimierungen

Innerhalb der Optimierungen [WG84, WM96, Muc97, Mor98, Tra00, RBPL00, PLB+00], die von Übersetzern für imperative Sprachen durchgeführt werden, finden sich Ansätze [HPR89, CWZ90, HN90, LR92, CGS+99, BH99, Bla99, GS00a,

Tra00, WL02, Rit03], die sich mit der Analyse von Haldenoperationen und darauf aufbauenden Programmtransformationen befassen.

Haldenanalyse Eine Voraussetzung für das Verständnis darüber, wie innerhalb eines Programmablaufs mit den auf der Halde allozierten Objekten umgegangen wird, ist eine Programmanalyse der Referenzen und der referenzierten Objekte, eine *Haldenanalyse* [HPR89, CWZ90, HN90, LR92, Ste95, Tra00, WL02, HDH04, CGS+99, BH99, Bla99, GS00a, Tra00, CX03, Rit03]. Eine Haldenanalyse kann als eine Escape-Analyse ausgelegt sein [CGS+99, BH99, Bla99, GS00a, Tra00, CX03, Rit03], wie wir sie im nächsten Absatz vorstellen, oder ihre Ergebnisse können in diesem Sinne interpretiert werden [Ste95, Tra00, HDH04].

Escape-Analyse und Allokation innerhalb der Aufrufschachtel Durch die *Escape-Analyse* für imperative Programme [Dol97, CGS+99, BH99, Bla99, GS00a, Tra00, Rit03] kann teilweise verfolgt werden, in welchem Teil des Programmes Zugriffe auf ein bestimmtes, einzelnes Objekt stattfinden. Für den Fall, dass alle Zugriffe durch die Laufzeit einer Prozedur p eingefangen werden können, innerhalb derer das Objekt alloziert wird, kann das Objekt — anstatt auf der Halde alloziert zu werden — auf der Aufrufschachtel der Prozedur p angelegt werden.

In seiner Dissertation [Rit03] erwähnt Ritzau die Idee, dass ein Objekt in der Aufrufschachtel einer Prozedur eingelagert werden kann, die die Lebensdauer des Objektes umfasst, auch wenn die Prozedur nicht die Allokationsanweisung enthält, die das Objekt erstellt, ohne sie jedoch weiter zu verfolgen.

Die Allokation innerhalb einer Aufrufschachtel macht die Speicherverwaltung für das jeweilige Objekt überflüssig. Insbesondere entfallen sämtliche Berechnungen innerhalb der Freispeicherverwaltung, die sonst bei der Speicherallokation nötig gewesen wären. Für diejenigen Objekte, die innerhalb einer Aufrufschachtel angelegt werden, können sich dabei merkliche Laufzeitvorteile ergeben.

Die Bedingungen, unter denen ein Objekt auf der Aufrufschachtel angelegt werden kann, sind jedoch nur für einen kleinen Teil der Objekte erfüllt, so dass für den großen, verbleibenden Teil der Objekte eine andere Methode zur Speicherverwaltung gefunden werden muss.

Reduktion von Haldenzugriffen Optimierungen mit dem Ziel, die Anzahl von Zugriffen auf Objekte auf der Halde zu reduzieren, finden wir in den Ergebnissen von [Tra00] und als Ziel von [CX03]. Als Folge solcher Optimierungen können einzelne Objekte, die ursprünglich auf der Halde alloziert wurden, in der Aufrufschachtel angelegt werden. Objekte in der Aufrufschachtel wiederum können in ihre Felder

aufgebrochen werden (engl. *unboxing*). Die Felder des Objekts können offen in die Prozedurschachtel eingebaut werden.

Auch hier entfällt — wie beim Einbau von Objekten auf der Aufrufschachtel — für jedes eingelagerte Objekt die Allokation und folglich auch die Deallokation des Objektes. Allerdings sind auch hier die Bedingungen, unter denen ein Objekt aufgebrochen werden kann, so streng, dass nur ein kleiner Teil der Objekte aus der Speicherverwaltung herausgehalten werden kann.

Zusammenfassung Keine der in diesem Abschnitt vorgestellten Techniken kann das Problem der Speicherverwaltung in imperativen Sprachen insgesamt lösen, oder bietet auch nur den Ausblick auf eine solche Lösung. Selbstverständlich wird man in einem Übersetzer auf keine dieser Optimierungen verzichten wollen, da für jedes Objekt, das nicht auf der Halde alloziert wird, bereits der Laufzeitbedarf des Programmes reduziert wird.

3.9 Zusammenfassung der verwandten Arbeiten

Echtzeitsysteme Wir haben in Abschnitt 3.1 einen Überblick über Echtzeitsysteme gegeben. Wichtig dabei ist die Abgrenzung zu den sehr unscharf definierten „weichen Echtzeitsystemen“. In Abschnitt 3.1.2 haben wir betont, dass bei Echtzeitsystemen nicht die mittlere, sondern immer nur die maximale Laufzeit entscheidend ist.

Speicherorganisation und manuelle Speicherverwaltung Nach dem grundlegenden Überblick über die Speicherarten und die Speicherorganisation in Abschnitten 3.2 und 3.3 haben wir uns den heute existierenden Ansätzen und Ideen zur Verwaltung des Haldenspeichers gewidmet. Wir haben mit der expliziten Speicherverwaltung eine häufig anzutreffende Speicherverwaltung beschrieben, die zwar eine transparente Laufzeit hat, die jedoch sehr arbeitsintensiv und fehleranfällig ist, so dass sie für Software, die in sicherheitskritischen Bereichen eingesetzt werden soll, ein kaum verantwortbares Risiko darstellt.

Speicherbereinigung In Abschnitt 3.5 haben wir die automatische Speicherbereinigung kennen gelernt, die durch ihre Konstruktion bereits Speicherzugriffsfehler vermeidet. Wir haben aber auch herausgearbeitet, dass die Laufzeit für die Speicherbereinigung zwar häufig so über die Laufzeit des Programms verteilt werden kann, dass sie nur selten als tatsächliches Problem in Erscheinung tritt; wir haben jedoch am Ende von Abschnitt 3.5.3 auch festgestellt, dass lediglich die Arbeit für

die Traversierung verteilt wird, und dass nicht etwa freier Speicher während der Traversierung entdeckt wird. Dadurch muß für Speicherbereiniger, die in Echtzeitsystemen eingesetzt werden, die maximale und dabei sehr hohe Laufzeit angesetzt werden.

Funktionale Sprachen In Abschnitt 3.6 haben wir die automatische Speicherverwaltung behandelt, die in funktionalen Sprachen eingesetzt werden kann. Wir mussten aber auch zur Kenntnis nehmen, dass sich die dort existierenden Ansätze nicht auf imperative Sprachen übertragen lassen.

Referenzzähler und Optimierungen des Übersetzers Wir sind in den Abschitten 3.7 und 3.8 auf Techniken eingegangen, die zwar Teile der Speicherverwaltung übernehmen, die sich jedoch nicht aller Objekte annehmen können, die auf der Halde alloziert werden.

Zusammenfassung Die existierenden Ansätze sind für Echtzeitsysteme entweder wegen ihrer Fehleranfälligkeit ungeeignet, oder für ihre Laufzeit lässt sich nur eine sehr konservative Abschätzung angeben, oder sie können sich nicht aller Objekte annehmen, die auf der Halde alloziert werden.

Aus dieser Zusammenfassung ziehen wir im nächsten Kapitel unsere Schlüsse, und legen uns dann auf ein Ziel für diese Arbeit fest.

„Unfortunately, there is no method of garbage collection that is generally suitable to real-time applications. [...] A third method is to perform garbage collection periodically by a parallel process of lower priority. Provided the synchronization problems can be solved, this provides the least unsatisfactory solution for real-time use.“

Ada83 Design Rationale

Kapitel 4

Ansatz und Vorgehen

4.1 Zielsetzung

In dieser Arbeit entwickeln wir einen neuen Ansatz zur Verwaltung des Haldenspeichers. Wir analysieren, wie ein Programm die Objekte auf der Halde nutzt, und wir setzen in das Programm Anweisungen ein, die alle allozierten Objekte wieder löschen.

Diese Anweisungen setzen wir so ein, dass zum einen kein Objekt gelöscht wird, bevor das letzte mal darauf zugegriffen wurde; zum anderen setzen wir die Anweisungen so ein, dass alle Objekte frühestmöglich gelöscht werden.

Dadurch, dass wir die Analyse der Nutzung von Haldenobjekten automatisch durchführen, entlasten wir den Programmierer von der Pflicht, diese Analyse manuell durchzuführen. Wir schließen damit insbesondere aus, dass der Programmierer hierbei Fehler macht, wegen derer das Programm zur Laufzeit wegen Speicherfehlern abgebrochen werden müsste.

Die Anweisungen, die wir in das Programm einsetzen, stehen an festen, bekannten Stellen des Programms. Ihr Laufzeitbedarf ist transparent und leicht abschätzbar. Im Gegensatz zu den typischerweise sehr pessimistischen Abschätzungen, die man bei Speicherbereinigern machen muss, bieten wir Abschätzungen, die sehr viel näher an dem tatsächlichen Laufzeitbedarf liegen.

Wir analysieren das Programm mit einer Zeigeranalyse, die wir zu diesem Zwecke entwickelt haben. Wir beschreiben diese Analyse und wie wir ihre Ergebnisse interpretieren.

Die Ergebnisse, die unsere Programmanalyse liefert, enthalten eine Aussage darüber, wie ein Programm die Objekte auf der Halde nutzt. Wir nutzen diese Erkenntisse, um eine sichere und effiziente Verwaltung des Haldenspeichers automatisch zu berechnen und umzusetzen. Wir erklären in den folgenden Abschnitten im Detail, wie wir dies tun. In dem folgenden Kapitel zeigen wir, wie sich unser

Ansatz auf Testprogrammen verhält.

Weiteres Vorgehen: In Abschnitt 4.2 verschaffen wir uns einen klaren Begriff darüber, was man unter der *korrekten* Nutzung von Haldenobjekten versteht. Wir entwickeln einen Begriff der Fehlerfreiheit, und einen Begriff der Effizienz.

In Abschnitt 4.3 stellen wir vor, auf welcher Infrastruktur wir unseren Ansatz umsetzen.

In Abschnitt 4.4 beschreiben wir die Programmanalysen, die wir für unseren Ansatz entwickelt haben, im Detail.

In Abschnitt 4.5.2 gehen wir auf *Analysekontexte* ein, ein Begriff, den wir in Abschnitt 4.4 nur informell behandeln, der aber im Weiteren eine entscheidende Rolle spielt.

In Abschnitt 4.6 beschreiben wir, wie wir mit den in der Analyse gewonnenen Daten die Objekte einfangen, die das Programm zur Laufzeit alloziert, und mit welcher Methode wir sie wieder löschen.

4.2 Korrekter Umgang mit Haldenspeicher

Wenn wir uns vornehmen, in Programme Anweisungen einzusetzen, die die auf der Halde allozierten Objekte deallozieren sollen, müssen wir uns zunächst darüber im Klaren sein, welche Anforderungen an die Speicherverwaltung gestellt werden. In diesem Abschnitt grenzen wir ab, was wir unter einer *korrekten* Speicherverwaltung verstehen. Wir verwenden den Begriff der Korrektheit im Folgenden immer nur in diesem Sinne.

Zunächst betrachten wir ein beliebiges Programm $\mathcal{P}$, das Objekte auf der Halde alloziert und wieder explizit dealloziert. Wir halten fest, wann wir einen einzelnen Lauf von $\mathcal{P}$ als korrekt im Sinne der Haldenspeicherverwaltung ansehen:

Definition 4.1 (Korrekter Lauf eines Programms $\mathcal{P}$) *Ein Lauf eines Programms $\mathcal{P}$ ist dann korrekt, wenn für jedes Objekt o, das $\mathcal{P}$ auf der Halde anlegt, folgendes* Protokoll für die Nutzung von Haldenobjekten *eingehalten wird:*

1. *Objekt o wird alloziert.*

2. *Auf o wird zugegriffen.*

3. *Objekt o wird dealloziert.*

4. *Auf o wird* nicht mehr *zugegriffen.*

Für jedes Programm $\mathcal{P}$, das den Haldenspeicher explizit verwaltet, ist es natürlich erforderlich, dass jeder Lauf, den es durchführt, korrekt im Sinne der Definition 4.1 ist.

Um sicherstellen zu können, dass jeder Lauf von $\mathcal{P}$ im o.g. Sinne korrekt ist, übertragen wir Definition 4.1 von der Laufzeit auf das Programm $\mathcal{P}$ selber:

Definition 4.2 (Korrekte Speicherverwaltung eines Programmes $\mathcal{P}$) *Wir sagen, dass ein Programm $\mathcal{P}$ den Haldenspeicher* korrekt nutzt, *wenn jeder Lauf, den $\mathcal{P}$ durchführen kann, korrekt im Sinne von Definition 4.1 ist.*

In Definition 4.2 haben wir nicht berücksichtigt, wie ihre Einhaltung für ein gegebenes $\mathcal{P}$ entschieden werden kann. Wollte man mit einer Programmanalyse überprüfen, ob Definition 4.2 für ein gegebenes $\mathcal{P}$ erfüllt ist oder nicht, so müsste man zur Kenntnis nehmen, dass es nicht möglich ist, das Verhalten von $\mathcal{P}$ präzise zu analysieren; eine Programmanalyse kann immer nur eine konservative Abschätzung liefern, die eine Obermenge des tatsächlichen Verhaltens von $\mathcal{P}$ zur Laufzeit ist. Es existieren also Programme, die zwar in jedem Lauf korrekt im Sinne von Definition 4.1 sind, für die wir jedoch nicht automatisch nachweisen können, dass sie korrekt im Sinne von Definition 4.2 sind. Da wir unseren Ansatz vollautomatisch durchführen wollen, müssen wir wir darauf verzichten, solche Programme als Ergebnis zu erstellen oder zu akzeptieren.

Nachdem wir angegeben haben, was wir unter einem korrekten Programm verstehen, klassifizieren wir diejenigen Verhaltensweisen, die gegen die Korrektheit verstoßen.

Zunächst betrachten wir Programme, die für einige Objekte möglicherweise den Schritt 4 aus Definition 4.1 nicht beachten:

Definition 4.3 (Fehlzugriff eines Programmes $\mathcal{P}$) *Wenn ein Programm $\mathcal{P}$ den Schritt 4 aus Definition 4.1 nicht beachtet, sagen wir, das $\mathcal{P}$ einen* Fehlzugriff *(engl.: stale pointer access) durchführt.*

Greift $\mathcal{P}$ auf ein Objekt o zu, das es bereits an die Freispeicherverwaltung zurückgegeben hat, so ist nicht auszuschließen, dass diese mit dem Speicher, der für o verwendet wurde, bereits eine andere Speicheranforderung beantwortet hat, wobei der Speicher gemäß seiner neuen Verwendung mit Daten überschrieben wurde. In diesem Fall enthält er also nicht mehr die Daten, die den Zustand von o repräsentieren.

Nun betrachten wir Programme, die möglicherweise den Schritt 3 (und, zwangsläufig, Schritt 4) aus Definition 4.1 nicht beachten:

Definition 4.4 (Speicherleck eines Programmes $\mathcal{P}$) *Wenn ein Programm $\mathcal{P}$ für ein Objekt o, das es auf der Halde alloziert hat, Schritt 3 nicht durchführt, so sagen wir, dass $\mathcal{P}$ ein* Speicherleck *hat.*

Wenn $\mathcal{P}$ während eines Laufes Objekte alloziert, die nie wieder dealloziert werden, so besteht die Gefahr, dass der Speicherverbrauch über jede feste Grenze steigt, so dass möglicherweise Läufe von $\mathcal{P}$ wegen Speichermangels abgebrochen werden müssen.

Bisher haben wir lediglich verlangt, das alle Objekte, die alloziert werden, auch wieder dealloziert werden. Von einem Programm wird man aber auch erwarten, dass es mit dem Haldenspeicher — über den Lauf hinweg — sparsam umgeht, und ihn effizient ausnutzt. Wir suchen dies zu erfassen in der folgenden Definition:

Definition 4.5 (Effiziente Speichernutzung eines Programmes $\mathcal{P}$) *Wir sagen, dass ein Programm $\mathcal{P}$ den Haldenspeicher* effizient *nutzt, wenn jedes Objekt o, das alloziert wird, frühestmöglich wieder dealloziert wird.*

Diese Definition lässt leider einen gewissen Interpretationsspielraum. Für ein einzelnes Objekt o, das in einem Lauf von $\mathcal{P}$ alloziert wird, existiert ein Zeitpunkt, zu dem $\mathcal{P}$ zum letzten Mal auf o zugreift. Mit Definition 4.5 ist nun nicht gemeint, dass die allernächste Aktion von $\mathcal{P}$ darin bestehen muss, dass o dealloziert wird. In manchen Fällen kann bei einem gegebenen Zugriff auf ein Objekt o gar nicht entschieden werden, dass es sich bei diesem Zugriff um den *letzten* Zugriff handelt, den das Programm auf o durchführen wird. In anderen Fällen ist es häufig wünschenswert, die Deallokation eines Objektes hinauszuzögern, weil die Deallokation zu einem späteren Zeitpunkt effizienter erfolgen kann, oder weil die Platzierung der Anweisungen, die die Deallokation vornehmen, so besser zur inneren Struktur der Software passt. Um auch diese Aspekte genügend zu berücksichtigen, verstehen wir Definition 4.5 so, dass Speicher nicht ohne besonderen Grund länger alloziert bleiben soll, als dies nötig ist.

4.3 Programmrepräsentation

4.3.1 Übersetzerstruktur

Der Übersetzer, den wir in unserer Arbeit verwenden, ist ein statischer Übersetzer [WG84, GE90, WM96, Muc97, Mor98, PLB+00, RBPL00], der Quelldateien in Assemblercode übersetzt. Der Assemblercode wird mit den plattformspezifischen Bindern (*linker*) zu einem ausführbaren Programm (*executable*) übersetzt.

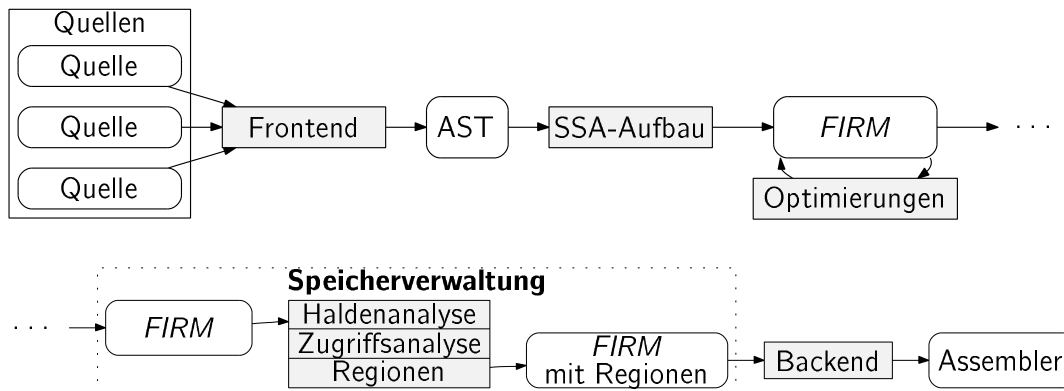


Abbildung 4.1: Struktur des Übersetzers

Die Struktur des Übersetzers ist in Abbildung 4.1 dargestellt[1]. Die Quelldateien werden im *Frontend* eingelesen und syntaktisch und semantisch analysiert. Das Ergebnis der semantischen Analyse steht als attributierter Strukturbaum (engl. *abstract syntax tree*, AST) zur Verfügung. Aus dem Strukturbaum bauen wir dann unsere Zwischenrepräsentation *Firm* auf, die sowohl die statischen Strukturen (Datentypen, Klassen und Methoden) als auch die Anweisungen der Methoden enthält. Die Effekte von externen Bibliotheken, die nicht in Quelle zur Verfügung stehen, stellen wir dem Übersetzer in einer abstrakten Beschreibung zur Verfügung. Daraus werden ebenfalls *Firm*-Graphen aufgebaut. Diese Graphen werden so beschrieben und aufgebaut, dass sie in Programmanalysen anstelle des (nicht vorhandenen) Quellcodes von Bibliotheksfunktionen analysiert werden können.

Auf *Firm* werden einige klassische Optimierungen ausgeführt [CC95, Tra00]. In dem als „Speicherverwaltung" markierten Bereich — zwischen *Firm* und *Firm mit Regionen* — setzen wir unseren Ansatz ein; die dort aufgeführten Begriffe erklären wir im Laufe dieses Kapitels. Das Ergebnis wird wieder in *Firm* dargestellt, so dass es als Eingabe des *Backends* [Boe05] verwendet werden kann.

4.3.2 Statische Einmalzuweisung

Wir führen alle Arbeiten, die im Folgenden beschrieben werden, auf unserer internen Zwischenrepräsentation *Firm* [TLB99] durch. In diesem Abschnitt stellen wir diese Zwischenrepräsentation vor. Für Details verweisen wir auf den technischen Be-

[1] Wir erlauben uns, für einige der Komponenten die gebräuchlichen englischen Begriffe zu verwenden.

richt [TLB99]. Die Zwischenrepräsentation *Firm* basiert auf dem Prinzip der Statischen Einmalzuweisung (engl. *static single assignment*, SSA) [CFR+91, CP93, Cli93, PL94, BM94, BHS95, CC95, Muc97, BCHS97, Mor98, App98, TLB99, Bla99, KCL+99, AH00].

Firm wurde als SSA-basierte Zwischenrepräsentation entwickelt, um als Grundlage von optimierenden Übersetzern [Tra00] und Softwareanalysesystemen zu dienen. So ist sie für unsere Zwecke geeignet. Im Folgenden geben wir eine kurze Beschreibung von *Firm*, soweit sie für die weiteren Ausführungen nötig ist. Für Details verweisen wir auf den Technischen Bericht [TLB99].

Definition von SSA Mit SSA werden die Anweisungen einer einzelnen Prozedur dargestellt. Von gängigen Programmiersprachen kennen wir das Konzept von *lokalen Variablen*, die während der Ausführung der Prozedur beschrieben und gelesen werden können.

Für Programme, die in SSA-Form sind, gilt für diese Variablen eine besondere Eigenschaft:

Definition 4.6 (SSA-Eigenschaft) *Eine Prozedur ist dann in* SSA-*Darstellung, wenn gilt:*

1. *Auf jede lokale Variable wird* **genau einmal** *schreibend zugegriffen. Dieser schreibende Zugriff dominiert alle lesenden Zugriffe auf die Variable.*

2. *Hängt ein Wert von der Ablaufsteuerung ab, so wird er durch eine* Φ*-Funktion definiert.*

Eine Φ-Funktion, die mehrere Definitionen zusammenfasst, steht an der *Dominanzgrenze* [CFR+91, BM94, BHS95, BCHS97, AH00] der Definitionen. Sie hat so viele Argumente, wie der Grundblock, in dem sie steht, Ablaufvorgänger hat. Wird der Block über den k-ten Ablaufvorgänger betreten, wählt die Φ-Funktion ihr k-tes Argument aus. Alle Φ-Funktionen, die in einem Grundblock stehen, werden gleichzeitig und als erste Anweisung des Grundblockes ausgeführt.

SSA als Graph Die Variablen einer Prozedur in SSA-Form haben eine Eigenschaft, die wir bereits bei den funktionalen Sprachen kennengelernt haben [App98]: Jede Variable wird genau einmal definiert, und ändert ihren Wert im Folgenden nicht mehr. Betrachten wir für eine feste Variable x ihre Definition und alle ihre Benutzungen, so stellen wir fest, dass x selber mit ihrem definierenden Ausdruck *expr* identifiziert werden kann, und an ihren Benutzungen lediglich eine Referenz auf *expr* ist.

		Erste Operation	
		schreiben	lesen
Zweite	schreiben	Ausgabeabhängigkeit	Gegenabhängigkeit
Operation	lesen	Eingabeabhängigkeit	(Pseudoabhängigkeit)

Tabelle 4.1: Abhängigkeiten zwischen Haldenoperationen [Tra00]

Die Nutzung einer Variablen x, die durch einen Ausdruck *expr* definiert ist, können wir durch eine Kante von Nutzung nach *expr* ersetzen. Die Variable selber wird dabei überflüssig[2]. Mit dieser Erkenntnis [Cli93] können wir Prozeduren in SSA-Form mit einem Graphen darstellen.

Abhängigkeiten zwischen Haldenoperationen Im vorigen Absatz haben wir den Übergang von einem AST, der eine Prozedur in SSA-Form darstellt, zu dem SSA-Graphen, in dem Operationen durch Kanten verbunden sind, beschrieben. Mit den Kanten zwischen den Operationen sind genau die Abhängigkeiten dargestellt, die zwischen den Operationen bestehen. Betrachten wir jedoch diejenigen Operationen, die auf Objekte auf der Halde lesend und schreibend zugreifen, so sehen wir, dass zwischen diesen Operationen Abhängigkeiten bestehen können, die wir mit der bisher dargestellten Form des SSA-Graphen nicht ausdrücken. Zwischen Operationen, die möglicherweise die gleiche Stelle der Halde lesen oder beschreiben, besteht eine Abhängigkeit über den Zustand der Halde selber. Die Abhängigkeiten, die bestehen können, haben wir in Tabelle 4.1 zusammengefasst. Mit Ausnahme der Kombination von zwei lesenden Operationen ist die Ausführungsreihenfolge zweier Haldenoperationen durch die genannten Abhängigkeiten entscheidend. In der bisher gelieferten Graph-basierten SSA-Darstellung müssen wir also eine Möglichkeit vorsehen, die Abhängigkeit von Haldenoperationen über den Zustand der Halde darzustellen.

Haldenoperationen in SSA-Graphen Um die Abhängigkeiten zwischen Haldenoperationen in SSA-Graphen auszudrücken, behandeln wir den Zustand des Haldenspeichers wie eine lokale Variable. Jede Haldenoperation liest den Wert dieser Variable, und definiert die Variable mit einem neuen Wert. Wie für die anderen lokalen Variablen setzen wir für den Zustand des Haldenspeichers Φ-Funktionen wie in Definition 4.6 beschrieben ein.

Um mehr Details über die *Firm*-Graphen beschreiben zu können, widmen wir uns zunächst in Abschnitt 4.3.3 der Repräsentation der statischen Programmteile,

[2] Der ursprüngliche Name der Variable kann von Interesse sein, wenn man zusätzlich noch Informationen zur Fehlersuche am laufenden Programm (engl. *debugging*) erhalten möchte.

um dann in Abschnitt 4.3.4 auf die SSA-basierte Repräsentation der Prozeduren einzugehen.

4.3.3 Statische Programmstruktur

Das Ziel, das mit der Entwicklung von *Firm* erreicht wurde, ist die Darstellung des gesamten Programmes innerhalb einer Infrastruktur. Innerhalb von *Firm* sind alle Informationen repräsentiert, die von dem Übersetzer aus den Quelldateien gelesen und die durch das Backend übersetzt werden. Durch *Firm* wird immer genau ein vollständiges Programm $\mathcal{P}$ dargestellt. Welche Elemente dazu nötig sind, geben wir im Folgenden an:

Prozeduren Es existiert eine Menge von Prozeduren P, die alle als SSA-Graphen repräsentiert sind. Eine Prozedur $p \in P$ bekommt zu Beginn ihrer Ausführung Null oder mehr atomare Werte übergeben, und gibt Null oder einen atomaren Wert zurück. Die Tupel bestehend aus dem Datentyp des Rückgabewertes und der zulässigen Datentypen der Argumente der Prozedur nennen wir ihre Signatur $\mathit{signature}(p)$. In *Firm* werden Parameter immer per Wert übergeben. Andere Methoden der Parameterübergabe, wie z. B. durch Namens- und Referenzparameter bilden wir auf die Werteparameter von *Firm* ab.

Wir nutzen Prozeduren in *Firm*, um einige Prozeduren des Quellprogramms umzusetzen. Wir setzen die sog. statischen Methoden von *Java* und die Konstruktoren von Klassen um [GJS97, § 8.4.3.2 und § 8.8], und wir können damit die gleich benannten Konzepte aus C++ [Koe97, § 9.4 bzw. § 12.1] und die sog. freien Funktionen [Koe97, § 8.3.5] umsetzen.

Aufrufklammern Die Modi der Parameter einer Prozedur bestimmen implizit, wie die Aufrufklammer der Prozedur zusammengesetzt sein muss. Solange die zu Grunde liegende Quellsprache ausschließlich Parameterübergabe per Wert- oder per Referenzübergabe vorsieht, ist dies ausreichend. Um eine Parameterübergabe mit Namensparametern oder mit Namens/Ergebnisparametern umzusetzen, oder um Zugriff auf die dynamischen Vorgänger in der Aufrufhierarchie zu ermöglichen, können wir Aufrufklammern formell durch Objekte darstellen, die zwischen Prozeduren übergeben werden. Die Objekte, die wir aus diesem Grund erstellen, sind markiert, so dass das *Backend* sie wieder erkennen und geeignet behandeln kann.

Programmeinsprung In einem *Firm*-Programm existiert genau eine Prozedur, die als Programmeinsprungsprozedur ausgezeichnet ist. Die Ausführung des Pro-

gramms geschieht dadurch, dass diese ausgezeichnete Prozedur ausgeführt wird. Wir geben dieser Prozedur im Folgenden den Namen *main*.

Initiale Belegung von Variablen In *Firm* existieren keine impliziten Annahmen über initiale Werte von Variablen. Wir unterscheiden hier zwischen lokalen Variablen auf der Aufrufklammer, und globalen Variablen und Feldern von Objekten.

Lokale Variablen werden in *Firm*-Graphen nur implizit durch Datenflußkanten dargestellt (siehe dazu nachfolgenden Abschnitt 4.3.4), und initiale Werte werden dort durch Konstanten dargestellt werden. Findet in der Quellsprache auf eine lokale Variable zuerst ein lesender Zugriff statt, so folgen wir den Regelungen, die die Quellsprache für diesen Fall vorsieht und generieren eine Konstante mit dem vorgeschriebenen Wert. Liefert die Quellsprache hier keine Vorschrift, so erstellen wir eine Konstante, deren Wert als „unbekannt“ gesetzt wird.

Für die Werte, die Felder von Objekten direkt nach der Allokation haben, folgen wir der Quellsprache dadurch, dass wir Anweisungen generieren, die die geforderten Werte explizit schreiben. Für den Fall, dass wir die Initialisierung im *Backend* effizienter als durch Ausführung der generierten Anweisungen umsetzen können, werden diese Anweisungen mit einer Markierung versehen, anhand derer sie das *Backend* erkennen kann.

Globale Variable müssen ebenfalls durch explizite Anweisungen initialisiert werden. Gegebenenfalls muss der Übersetzer eine eigene Prozedur generieren, die diese Initialisierung übernimmt. Zweckmässigerweise erstellt man aus der Prozedur, die in der Quellsprache als Programmeinsprungsprozedur erscheint, eine *Firm*-Prozedur, und generiert eine *main*-Prozedur in *Firm*, die zunächst die Initialisierungsprozedur und dann die Programmeinsprungsprozedur des Quelltextes aufruft.

Datentypen In *Firm* existieren Entsprechungen für alle Datentypen, die für das Programm $\mathcal{P}$ benötigt werden. Für atomare Werte, wie Gleit- und Festkommazahlen, für Bitmasken und für Zeiger existieren in *Firm* die sog. *Modi*. Die Verbundtypen werden durch *Entitäten* repräsentiert.

Modi Innerhalb des *Firm*-Graphen werden über die Kanten ausschließlich atomare Werte übertragen. Für diese atomaren Werte existieren in *Firm* sog. *Modi*. Es existieren konkrete und symbolische Modi. Wir stellen zunächst die Modi in ihrer Gesamtheit vor, erklären ihre Bedeutung später in Zusammenhang mit den Operationen, die sie nutzen.

Name	**Symbol**	**Bedeutung**
Ausführung	X	Berechtigt einen Grundblock zur Ausführung
Speicher	M	Zustand des Haldenspeichers
Boolescher Wert	B	$\{true, false\}$
Ganzzahl	I	in geeigneter Ausprägung
Gleitkommazahl	F	in geeigneter Ausprägung
Zahl	Z	F oder I
Zeiger	P	
Referenz	R	
Wert allg.	V	Z, B oder R
Ungültig	$\perp$	künstl. Typ für Prozeduren ohne Rückgabewert

Für die Ganzzahl- und Gleitkommazahl-Modi können wir Ausprägungen bilden, die sich in der Darstellung und Wortbreite unterscheiden.

Referenzen und Zeiger Innerhalb von *Firm* unterscheiden wir die Modi R für Referenzen und Z für Zeiger. Referenzen sind lediglich Deskriptoren für (oder Verweise auf) Objekte. Hinter Zeigern verbergen sich Speicheradressen. Dies beeinflusst, welche Operationen auf Werten dieser Modi zulässig sind: Auf Referenzen können wir immer die Feld- und Methodenauswahl durch den später beschriebenen *select*-Knoten durchführen, während arithmetische Operationen nicht definiert sind; mit Zeigern können wir arithmetische Operationen durchführen (Speicherarithmetik), während einer *select*-Operation mit einem Zeiger als Argument keine sinnvolle Semantik zugeordnet werden kann. Mit dieser Unterscheidung erhalten wir eine zusätzliche Möglichkeit, sowohl die *Firm*-Graphen selber als auch den Ablauf von Analysen auf *Firm*-Graphen auf Konsistenz zu überprüfen. Bei der Übersetzung von *Firm*-Graphen in Prozessor-spezifische Sprachen (*Assembler*) können Referenzen auf die am Ende von Abschnitt 4.3.5 erwähnten breiten Referenzen abgebildet werden, während Zeiger auf den auf den Adress-Datentyp der Zielarchitektur abgebildet wird, oder es werden beide Modi auf den dort vorhandenen Adress-Datentyp abgebildet.

Entitäten Verbundtypen werden in Entitäten durch Angabe der in ihnen enthaltenen Komponenten beschrieben. Der allgemeinste Verbundtyp ist der der *Klasse*. Andere Verbundtypen enthalten Teilmengen der Komponenten, die eine Klasse ausmachen können.

Die Unterscheidung zwischen Modi und Entitäten ist so gewählt, dass genau auf die Werte, die durch *Modi* beschrieben werden, atomar zugegriffen werden kann,

während aus *Entitäten* zunächst ihre Komponenten ausgewählt werden müssen. Anweisungen, die in einer Programmiersprache auf solchen Werten operieren, die in *Firm* durch *Entitäten* beschrieben werden, und die in der Programmiersprache als atomare Operation erscheinen, werden bei dem Aufbau von *Firm* in Operationen zerlegt, die auf den Komponenten operieren. Ein Beispiel hierfür ist die Zuweisung zwischen zwei Variablen in `C`, deren Typ ein `struct` ist.

Klassen Eine Klasse besteht aus Methoden und Feldern. Zur Laufzeit können durch das Programm Objekte alloziert werden, die *Exemplare* von Klassen sind. Auf die Klassen in *Firm* können wir Verbundtypen und Klassen von objektorientierten Sprachen abbilden,

Objekte und Exemplare Wir verwenden die Begriffe *Objekt einer Klasse* und *Exemplar einer Klasse* synonym. In den Fällen, in denen es uns darauf ankommt, dass ein Objekt o eine bestimmte Klasse D repräsentiert, reden wir von o als einem *Exemplar von* D. Wenn es nicht entscheidend ist, was für eine Klasse repräsentiert wird, reden wir von dem *Objekt* o.

Methoden und Felder Die Felder einer Klasse c beschreiben, welche Datenfelder in einem Objekt vorhanden sein müssen, das ein Exemplar von c ist. Jedes Feld hat einen atomaren Typ.

Die Methoden $methods(c)$ einer Klasse c sind Platzhalter für Aktionen, die mit Exemplare von c ausgeführt werden können. Wie bei Prozeduren existiert für jede Methode $m \in methods(c)$ eine Signatur $signature(m)$. Eine Methode $m \in methods(c)$ kann durch eine Prozedur $p \in P$ *implementiert* sein, die die gleiche Signatur wie m hat. Ist m durch p implementiert, dann bedeutet dies, dass der Aufruf von m eines Objekts von Klasse c die Ausführung der Prozedur p verursacht. Wir notieren dann: $p = impl(c, m,)$.

Abstrakte Methoden und Klassen Eine Methode, die zwar in einer Klasse spezifiziert ist, für die jedoch keine Prozedur angegeben ist, die diese Methode implementiert, nennen wir eine *abstrakte Methode*. Eine Klasse, die mindestens eine abstrakte Methode enthält, nennen wir eine *abstrakte Klasse*. Von einer abstrakten Klasse dürfen keine Objekte instantiiert werden.

Die Signatur einer Klasse Die Menge an Signaturen der Methoden einer Klasse, und die Typen der Felder der Klasse, nennen wir die Signatur der Klasse.

Vererbung und Polymorphie Klassen haben jeweils null oder mehr Unter- und Oberklassen. Hat Klasse c die Unterklasse c', so bedeutet dies gleichzeitig, dass c' die Oberklasse c hat. Ist entweder $c'' = c$ oder ist c' eine direkte Unterklasse von c, so notieren wir: $c' \to c$. Ist $c'' = c$ oder ist c'' eine transitive Unterklasse von c, so notieren wir: $c'' \to^+ c$. Der Graph $\{(c,c') \in C \times C : c \neq c' \wedge c' \to c\}$ muss stets azyklisch sein. Eine Klasse c *erbt* alle Merkmale ihrer Oberklassen. Für die Felder bedeutet dies, dass in c — zusätzlich zu den für c spezifizierten Feldern — auch alle Felder ihre Oberklassen vorhanden sind.

Für Methoden bedeutet dies Folgendes: Für jede Methode m, die in einer Klasse c existiert, existiert m auch in jedem $c' \to^+ c$, und

1. wenn m in c abstrakt ist, kann m in c' abstrakt sein, oder sie kann durch eine Prozedur implementiert sein;

2. wenn m in c durch eine Prozedur p implementiert ist, kann m in c' auch durch p oder durch eine andere Prozedur p' implementiert sein.

Ist m in c und c' unterschiedlich implementiert, so sagen wir, dass in Klasse c' die Methode *m überschrieben wird*.

Durch Vererbung erreichen wir Polymorphie: Da Exemplare einer Klasse $c' \to c$ alle Merkmale von Exemplaren von c haben, darf zur Laufzeit in jede Operation, die für Exemplare von c möglich ist, auch ein Exemplar von c' eingehen. Bei Methodenaufrufen wird dann die Prozedur ausgeführt, die die aufgerufene Methode in der Klasse c' implementiert. Ein Aufruf, der syntaktisch an die Methode m einer Klasse c geht, kann also zur Laufzeit an die Implementierung von m in einer beliebigen Unterklasse $c'' \to^+ c$ erfolgen. Die Menge aller Prozeduren, die hier ausgeführt werden können, ist also $impl^*(c,m) := \{p = impl(c'',m) | c'' = c \vee c'' \to^+ c\}$.

4.3.4 Firm-Graphen

Die Implementierung einer einzelnen Prozedur wird in *Firm*, wie bereits in Abschnitt 4.3.2 vorbereitet, als ein Graph dargestellt.

Die Knoten dieses Graphen sind Operationen; die Kanten zwischen den Knoten stellen die Verwendung von Werten dar. In *Firm* sind die Kanten immer vom Konsumenten eines Wertes zu dessen Produzenten gerichtet. Davon abgesehen ist ein *Firm*-Graph ein Datenflussgraph, dessen Knoten Operationen darstellen; eine Operation kann genau dann ausgeführt werden, wenn alle ihre Eingänge mit Daten versorgt sind. Nach der Ausführung produziert sie ein Ergebnis, das anderen Operationen zur Verfügung steht.

Konkrete und symbolische Knoten Die Knoten haben zum Teil ihre Entsprechung in der Quellsprache und auf dem Zielprozessor. Wir sprechen von *konkreten* Knoten. Andere Knoten haben keine solche direkte Entsprechung, sondern sie dienen zur (oder erleichtern die) Darstellung des SSA-Graphen; wir sprechen dann von *symbolischen* Knoten.

Grundblockgraph Aus den herkömmlichen Programmrepräsentationen [Mor98, RVW⁺97, WM96, WG84] übernommen haben wir die Grundblöcke und den Grundblockgraphen. Ein Grundblock ist ein Container, der andere Operationen enthält. Mit $BB(p)$ bezeichnen wir die Grundblöcke einer Prozedur $p \in P$.

Start- und Endblock Zwei Grundblöcke haben eine besondere Bedeutung im *Firm*-Graphen: Der Start- und der Endblock. Die Ausführung einer Prozedur beginnt damit, dass der Startblock ausgeführt wird. Die Ausführung des Endblockes symbolisiert das Ende der Prozedurausführung. Der Startblock hat als einziger Grundblock keine Vorgänger, und der Endblock darf nicht als Vorgänger eines Grundblockes auftreten.

Ablaufsteuerung Die Grundblöcke bilden mit ihren Operanden den Grundblockgraph. Die Operanden eines Grundblockes sind Operationen der Ablaufsteuerung. Die Operationen, die als Operanden eines Grundblocks b dienen, liegen in den Grundblöcken, die Ablaufvorgänger von b sind.

Es gibt zwei Knoten, die als Vorgänger von Grundblöcken auftreten können, den bedingten Sprung, und den unbedingten Sprung:

Name	Operation	Operanden und Modi	
bedingter Sprung	$cond(b) \to (x_t, x_f)$	b	B
unbedingter Sprung	$jmp() \to x$	x, x_t, x_f	X

Beide Operationen werden innerhalb eines Grundblockes immer zuletzt ausgeführt. Beide produzieren Werte vom symbolischen Typ X (für engl. *execute*), die es dem Ablaufnachfolger erlauben, die Ausführung seiner Operationen anzustoßen. Der unbedingte Sprung produziert immer einen Wert für den Ablaufnachfolger. Von den beiden Werten, die bei dem bedingten Sprung angegeben sind, wird immer nur ein Wert überhaupt produziert. Welcher dies ist, hängt von der Bedingung b ab: Ist sie wahr, so wird der Wert x_t produziert, sonst der Wert x_f. Der Modus von b ist der boolesche Modus B.

Es gibt noch zwei weitere Knoten, die wir zur Ablaufsteuerung verwenden: Den Start- und den Returnknoten.

Der Startknoten steht immer im Startblock, und symbolisiert den initialen Zustand bei Beginn der Prozedurausführung. Er hat selber keine Operanden. Die Wer-

te, die er produziert, hängen von der Signatur der Prozedur ab: Er produziert in jedem Fall den initialen Speicherzustand m vom Modus M und die initiale Ausführungsberechtigung, die, wie die der Wert von Sprüngen vom Modus X ist. Für jeden Parameter der Prozedur produziert der Startknoten einen Wert p_i, der den Modus' V_i des i-ten Parameters hat, und der den aktuellen Wert dieses Parameters repräsentiert.

Der Returnknoten hat einen oder zwei Operanden: Er konsumiert in jedem Fall den Speicherzustand m. Sofern die Prozedur einen Rückgabewert liefert, konsumiert er einen weiteren Operanden r mit dem Modus R des Rückgabewertes.

Name	**Operation**	**Operanden und Modi**	
Prozedurstart	$start() \rightarrow (m, x, p_0, \ldots, p_n)$	m	M
		x	X
Return ohne Wert	$return(m) \rightarrow x$	p_i	V_i
Return mit Wert	$return(m, v) \rightarrow x$	v	V

Returnknoten liegen immer in den direkten Vorgängern des Endblocks. Der Endblock ist der einzige Block, der Returnknoten als Vorgänger hat.

Ablaufsteuerung zur Ausnahmebehandlung Um die Mechanismen zur Ausnahmebehandlung gängiger objektorientierter Sprachen repräsentieren zu können, existieren in *Firm* folgende Knoten:

Name	**Operation**	**Operanden und Modi**	
Explizite Auslösung	$throw() \rightarrow x$	x	X
Sprung wg. Ausnahme	$excjmp() \rightarrow x$		
Return wg. Ausnahme	$excreturn(m) \rightarrow x$	m	M

Der *throw*-Knoten stellt den expliziten Einsprung in die Ausnahmebehandlung dar (äquivalent zur `throw`-Anweisung von `C++` oder *Java*), mit *excjmp* stellen wir einen Sprung dar, der deshalb und dann durchgeführt wird, weil bzw. wenn eine Ausnahme aufgetreten ist und weiter verarbeitet wird. Mit dem *excreturn*-Knoten stellen wir die Beendigung einer Prozedur wegen einer nicht behandelten Ausnahme dar. Er liefert, wie der *return*-Knoten, den Speicherzustand m, der zum Ausführungszeitpunkt herrscht. Wie die *return*-Knoten sind die *excreturn*-Knoten immer Operanden des Endblocks.

Weitere Details werden nicht durch die hier genannten *Firm*-Knoten dargestellt. Die konkrete Definition der Ausnahmebehandlung in der Quellsprache, wie zum Beispiel die Identifikation der Ausnahme oder das Finden einer geeigneten Behandlungsroutine für die Ausnahme muss durch weitere, geeignete *Firm*-Knoten implementiert werden.

Projektions-Knoten Manche *Firm*-Knoten haben mehr als einen einzigen Rückgabewert. Während wir in dieser Darstellung den Rückgabewert jeweils als Tupel angeben, werden in der Implementierung von *Firm* die einzelnen Komponenten eines Tupels durch die Projektions-Knoten unterschieden. Ein Projektions-Knoten hat den Tupel-wertigen Knoten als Operanden, und ist mit der Komponente des Tupels markiert, die er aus dem Operanden auswählt.

Name	**Operation**	**Operanden und Modi**	
Projektion	$proj_i\,(m_1, m_2, \ldots, m_n) \to m_i$	m_k	beliebig für $k \leftarrow [1, \ldots n]$

Φ-Knoten Für die Darstellung von Φ-Funktionen in SSA existiert in *Firm* der Φ-Knoten. Die Anzahl seiner Operanden ist, wie in Definition 4.6 erklärt, gleich der Anzahl der Ablaufvorgänger des Grundblockes, in dem sich der Φ-Knoten befindet. Die Modi aller Operanden müssen übereinstimmen.

Name	**Operation**	**Operanden und Modi**	
Φ-Knoten	$\Phi\,(v_1, \ldots, v_n) \to v$	$v_1, \ldots, v_n, v$	V und M

Konstanten Um konstante Werte darzustellen, verwenden wir einen eigenen Knoten. In der Implementierung von *Firm* kann dieser Knoten sowohl mit einem konkreten Wert ausgerüstet werden, als auch mit einem symbolischen Verweis versehen werden. Diese Verweise können sich auf Größen beziehen, die erst im Laufe der Übersetzung — zum Beispiel bei der Berechnung der Speicherabbildung der Klassen — bekannt werden. Die symbolischen Verweise werden dann durch konkrete Werte ersetzt, wenn diese berechnet worden sind. Mit V bezeichnen wir den Modus der Konstante:

Name	**Operation**	**Operanden und Modi**	
Konstante	$const\,() \to v$	v	V

Arithmetische Operatoren Die Operatoren, die in gängigen Programmiersprachen zu finden sind, sind auch in *Firm* vorhanden. Die Grundrechenarten auf Fest- und Gleitkommawerten sowie Bit-orientierte Operationen stehen zur Verfügung. Für die vollständige Aufzählung verweisen wir auf [TLB99].

Name	**Operation**	**Operanden und Modi**	
Arithm. Op.	$arith\,(z_1, z_2) \to z$	z, z_1, z_2	Z
Bit. Op.	$bitop\,(v_1, v_2) \to v$	v, v_1, v_2	$V \backslash F$

Typumrechnung und -interpretation Mit dem *conv*-Konten können wir drei verwandte Operationen ausdrücken: Zunächst die dynamische Uminterpretation einer Referenz auf Objekte von einem Typ T in eine Referenz auf einen Typ $T' \neq T$,

dann die Umrechnung von Werten der Modi in V untereinander, wie zum Beispiel die Umrechnung eines Gleitkommawertes in einen Festkommawert, und schliesslich die statische Uminterpretation zwischen beliebigen Modi.

Die Uminterpretation einer Referenz ist innerhalb von *Firm* nicht nur an den Stellen nötig, an denen der Quelltext dies verlangt (*type cast*), sondern auch dann, wenn dies durch die Quellsprache (wie zum Beispiel in [ES90, Absch. 4.6]) gefordert wird. Es wird dem *Backend* überlassen, die nötigen Umrechnungen ([ES90, Absch. 10.1c ff.]) zu bestimmen und einzufügen.

Name	**Operation**	**Operanden und Modi**	
Dyn. Uminterpretation	$conv(r) \rightarrow r'$	r, r'	R
Wert-Umrechnung	$conv(v) \rightarrow v'$	v, v'	V
Stat. Uminterpretation	$conv(v) \rightarrow v'$	v, v'	$R \cup V$

Haldenoperationen Unter Haldenoperationen fassen wir alle Operationen zusammen, die direkt mit Zugriffen auf Objekte zu tun haben, die auf der Halde alloziert sind. Die unterschiedlichen Aufgaben, die dabei zu erfüllen sind, sind in *Firm* auf unterschiedliche Knoten aufgeteilt. Wir stellen zunächst die nötigen Knoten vor. Dabei sei f ein Feld eines Objektes, und v gleich dem Modus dieses Feldes.

Name	**Operation**	**Operanden und Modi**	
Feld auswählen	$select(m, r, f) \rightarrow (m', p)$	m, m'	M
Wert laden	$load(m, p) \rightarrow (m', v)$	r	R
Wert schreiben	$store(m, p, f) \rightarrow m'$	p	P
Objekt allozieren	$alloc(m) \rightarrow (m', r)$	v	V

Alle Haldenoperationen konsumieren und produzieren einen Wert vom Modus M. Dieser Wert stellt den Zustand des Hauptspeichers vor bzw. nach der Ausführung der Operation dar. Damit stellen wir die Abhängigkeiten zwischen den Operationen über den Inhalt der Halde dar.

Der Parameter r ist stets die Referenz auf das Objekt, dessen Feld f gelesen bzw. beschrieben werden soll. Der *select*-Knoten kapselt die Berechnung des Zeigers auf das Feld f aus der Referenz r. Der *alloc*-Knoten schließlich dient der Allokation von Objekten. Er wird mit einem symbolischen Verweis auf die Klasse ausgerüstet, von der ein Exemplar alloziert werden soll. Aus diesem Verweis wird im Backend des Übersetzers die benötigte Größe der Allokation berechnet.

Prozedur- und Methodenaufrufe Um Aufrufe an Prozeduren und Methoden auszudrücken, existieren in *Firm* die Knoten *select* und *call*. Es seien $v_1, \ldots, v_n$ die Modi der Parameter der Prozedur bzw. der Methode, und v_r der Modus des

Rückgabewertes; weiterhin sei x eine Prozedur, und y eine Methode:

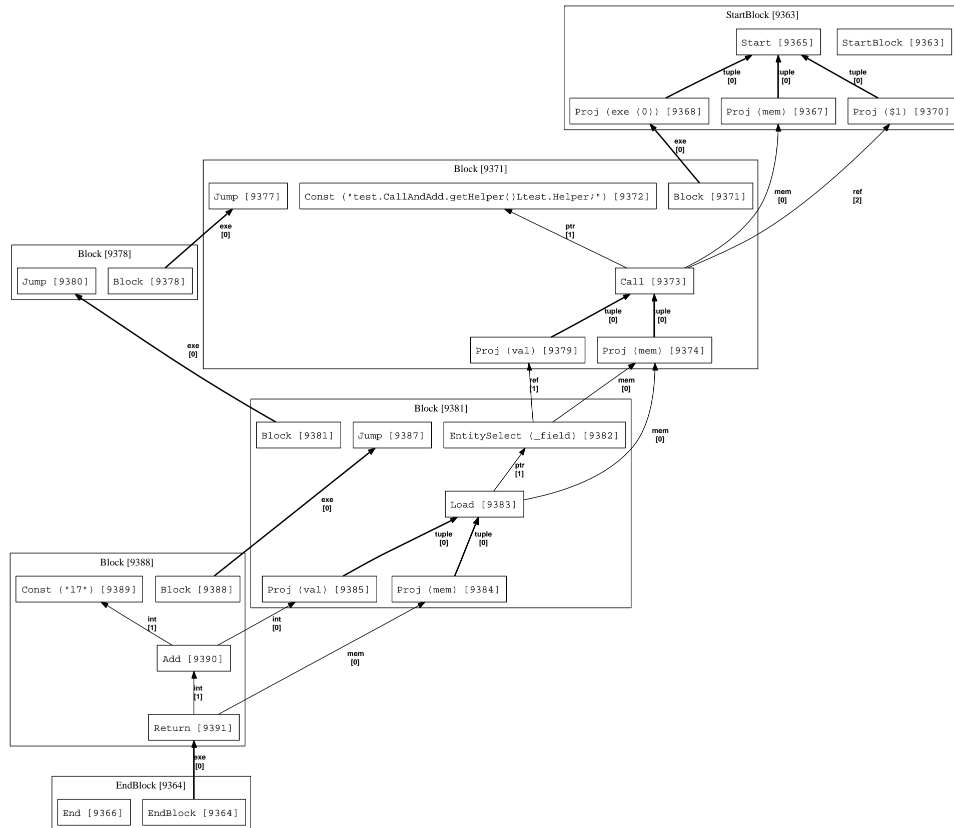
Name	Operation	Operanden und Modi	
Methodenauswahl	$select(m, r, x) \to (m', p)$	m, m'	M
Prozedurauswahl	$select(m, \perp, y) \to (m', p')$	$v_1, \ldots, v_n$	$V_1, \ldots, V_n$
Methode aufrufen	$call(m, p, v_1, \ldots, v_n) \to (m', v_r)$	v_r	V_r
Prozedur aufrufen	$call(m, p, v_1, \ldots, v_n) \to (m', v_r)$	r	R
		p	P

Wie bei den Haldenoperationen kapselt der *select*-Knoten die Berechnung eines Deskriptors für die aufzurufende Prozedur bzw. das Nachschlagen der Prozedur, die in der Klasse des Objektes r als Implementierung der aufgerufenen Methode steht. Der *call*-Knoten stellt den eigentlichen Aufruf der ausgewählten Prozedur dar.

Konventionen für Methodenaufrufe Methoden sind — im Gegensatz zu Prozeduren — an Klassen gebunden. Sie stellen eine Abstraktion von Prozeduren dar, bei der die Zugehörigkeit der Methode zur Klasse betont wird, während durch Polymorphie von der Implementierung abstrahiert wird. Methoden werden immer über ein Exemplar einer Klasse aufgerufen. Der Ablauf der Methode bezieht sich primär auf dieses Exemplar. Während der Ausführung der Methode (das heisst natürlich, der Prozedur, die die Methode implementiert), steht ein Verweis auf das Exemplar zur Verfügung. Er tritt als „`this`" in `C++` [ES90] und *Java* [GJS97] und als „`self`" in *Sather* [Goo96, BGK99] und *Eiffel* [Mey92] auf.

In *Firm* muss diese Referenz, wenn sie gebraucht wird, explizit in die Signatur von Methode und implementierenden Prozeduren aufgenommen und beim Methodenaufruf als Argument übergeben werden. Per Konvention wird die Referenz als nullter Parameter übergeben; danach folgen die Parameter, die sich aus der Deklaration der Methode im Quellcode ergeben.

Strikte Ausführung von Firm-Graphen Die Ausführung von *Firm*-Graphen ist, abgesehen von den im Folgenden genannten Ausnahmen, strikt: Wird ein Grundblock ausgeführt, so müssen alle in ihm enthaltenen Operationen ausgeführt werden. Ausnahmen hierbei sind die *proj*-Knoten, die den nicht genommenen Ausgang eines bedingten Sprunges auswählen, und *excjmp*-Knoten, in dem Fall, dass keine Ausnahme aufgetreten ist. Mit der Ausführung eines Grundblockes, der nicht der Endblock ist, muss auch dessen Nachfolger ausgeführt werden. Wird keines der Ergebnisse, das eine Operation produziert, von einer anderen Operation verwendet, so braucht diese Operation nicht ausgeführt werden. Sie zählt dann auch selber nicht




```
class Helper {
  public int _field;
}
public class CallAndAdd {
    public int call_and_add (CallAndAdd caa) {
       int num = caa.getHelper ()._field;
       num += 17;
       return (num);
     }
    private Helper getHelper () {
      return (_helper);
    }
    private Helper _helper;
  }
}
```

Abbildung 4.2: Beispiel für einen *Firm*-Graphen. Dargestellt ist die Methode `CallAndAdd.call_and_add ()`. Der *Java*-Quellcode, aus dem dieser Graph entstand, ist darunter angegeben.

mehr als Konsument ihrer Operanden, wodurch ggf. weitere Operationen nicht ausgeführt zu werden brauchen.

4.3.5 Speicheranordnung und Polymorphie

In diesem Abschnitt erwähnen wir einige Details der Umsetzung der Zwischenrepräsentation *Firm* in Assembler. Um die Übersetzung von *Firm*-Graphen in Assembler brauchen wir uns in dieser Arbeit nicht zu sorgen[Boe05], aber wir gehen im Folgenden auf die Umsetzung der statischen Strukturen ein, die in einem *Firm*-Programm spezifiziert sind.

Speicheranordnung von Objekten Die Felder, die eine Klasse enthält, werden nacheinander im Speicher angeordnet. Um Reihungen (engl. *arrays*) zu speichern, können die einzelnen Einträge aneinandergereiht werden.

Felder können nicht immer lückenlos aneinandergereiht werden. Von wenigen Ausnahmen abgesehen verlangen alle Prozessorarchitekturen, dass jedes Feld mindestens nach seinem eigenen Umfang ausgerichtet ist, dass also ein Feld mit einer Länge von n Bytes an einer Position angeordnet wird, die ein ganzzahliges Vielfaches von n ist.

Um die Speicheranordnung für Exemplare einer Klasse c' zu berechnen, werden zunächst für jede Oberklasse c von c' alle Felder von c genau so angeordnet, wie für Exemplare von c. Danach werden die Felder angereiht, die in c' spezifiziert sind.

Um Eigenschaften wie die sog. virtuellen Oberklassen von `C++` zu implementieren, können wir die in [Tra00, ES90] beschriebenen Techniken verwenden.

Implementierung von polymorphen Methodenaufrufen Um den polymorphen Aufruf von Methoden einem Exemplar von Klasse c zur Laufzeit auf diejenigen Prozeduren weiterzuleiten, mit der c die Methoden jeweils implementiert, wird eine Zuordnungstabelle (engl. *vtable*) pro Klasse generiert. In einer Tabelle für Klasse c sind Deskriptoren für die Prozeduren gespeichert, die Methoden von c implementieren. Der Zugriff auf die Tabelle geschieht über einen Methoden-Index. Der Index für eine Methode ist für alle Klassen, die diese Methode über Vererbung gemeinsam haben, gleich. Das heißt, dass für eine Methode m mit dem Index i_m in jeder Zuordnungstabelle der Deskriptor für die jeweils implementierende Prozedur an Stelle i_m eingetragen ist.

Wenn die zu Grunde liegende Quellsprache Mehrfachvererbung vorsieht, kann es nötig sein, dass der Zeiger auf das Exemplar vor dem Aufruf angepasst werden muss. Dies ist immer dann nötig, wenn die aufgerufene Methode in einer Oberklasse c^* von c implementiert ist, deren Felder nicht am Anfang des Datenbereiches

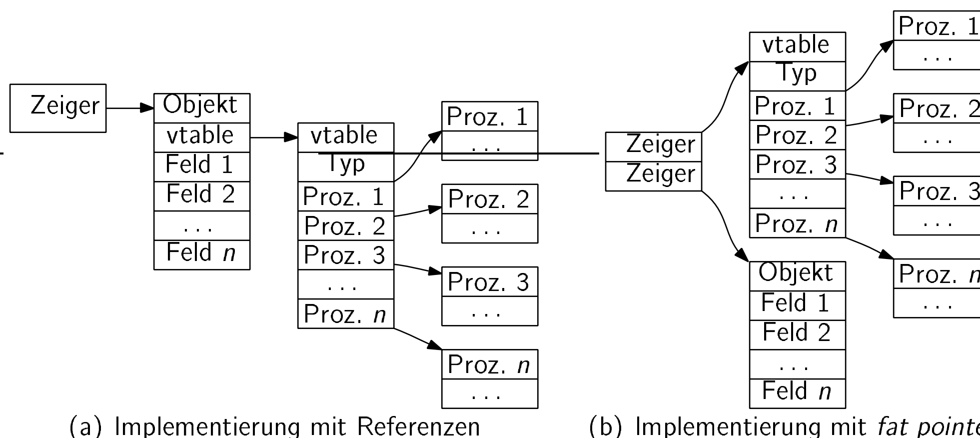
von c angeordnet wurden. Für eine ausführliche Beschreibung verweisen wir wieder auf [Tra00, ES90].

Bei einem polymorphen Methodenaufruf an ein Exemplar von Klasse c muss die Zuordnungstabelle für c gefunden werden. Dies kann man auf zwei Arten implementieren:

Verweis aus dem Exemplar: In jedem Exemplar wird ein Feld generiert, in dem ein Verweis auf die Zuordnungstabelle abgelegt wird. Um den Methodenaufruf durchzuführen, wird dieses Feld ausgelesen; dann wird mittels des Methoden-Indexes der gesuchte Prozedurdeskriptor nachgeschlagen. In Abbildung 4.3(a) ist dies an einem Beispiel dargestellt.

Breite Referenzen: Eine Referenz auf ein Exemplar besteht aus zwei Zeigern: Ein Zeiger, der auf den Beginn des Datenbereiches des Exemplares zeigt, und ein Zeiger, der auf die Zuordnungstabelle zeigt. Um den Methodenaufruf durchzuführen, wird über den Verweis auf die Zuordnungstabelle der Prozedurdeskriptor direkt ausgelesen. In der Literatur wird diese Technik mit dem Begriff „*fat pointer*“ bezeichnet. Sie ist in Abbildung 4.3(b) gezeigt.

Bei der Implementierung mit Zeigern muss also auf das Exemplar zugegriffen werden, um einen Methodenaufruf durchzuführen, während bei der Verwendung von *fat pointers* nur die Referenz auf das Exemplar benötigt wird.



(a) Implementierung mit Referenzen (b) Implementierung mit *fat pointers*

Abbildung 4.3: Implementierung von *fat pointers* und normalen Referenzen

4.4 Zeigeranalyse

Um Informationen darüber zu erhalten, wie ein Programm mit Referenzen umgeht, und wie es auf Objekte zugreift, verwenden wir eine Zeigeranalyse. Eine Zeigeranalyse ist eine Programmanalyse, die diejenigen Anweisungen analysiert, in denen Referenzen verwendet werden[3].

Wir geben zunächst einen kurzen Überblick über Zeigeranalysen im Allgemeinen und widmen uns dann unserer Implementierung einer Zeigeranalyse im Speziellen.

Abstrakte Werte In einer Programmanalyse werden die Werte, die Ausdrücke zur Laufzeit annehmen können, abstrahiert. Zur Abstraktion wird ein vollständiger Verband [NNH99] verwendet.

Die abstrakten Werte werden so gewählt, dass sie einen vollständigen Verband $(\mathcal{V}, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ bilden [BS81]. Dabei ist $(\mathcal{V}, \sqsubseteq)$ eine teilgeordnete Menge, und $\sqcap$ bzw. $\sqcup$ das Supremum bzw. das Infimum der Teilordnung; $\perp$ bzw. $\top$ sind das kleinste bzw. das grösste Element von $\mathcal{V}$. In Abschnitt 4.4.1 werden wir beschreiben, wie wir diese Werte für unsere Analyse wählen.

Kontexte Da eine Operation *op* an einer Stelle *s* im Programm auf mehr als einem einzigen Ausführungspfad erreicht werden kann, sucht man in einer Programmanalyse, diese unterschiedlichen Ausführungen zu unterscheiden. Diejenigen Ausführungspfade, die *nicht* unterschieden werden, fasst man jeweils in einem *Kontext* zusammen:

Definition 4.7 (Kontext) *Für eine Operation op an einer Stelle s im Programm verstehen wir unter einem* Kontext δ *von op eine Teilmenge aller Ablaufpfade, über die op bei s erreicht werden kann. Die Menge* $\Delta(op)$ aller Kontexte *von op ist eine Partition aller Ablaufpfade, auf denen op erreicht werden kann. Die Menge* Δ *ist die Menge aller Kontexte, die innerhalb eines Programms existieren.*

Transferfunktionen Eine Zeiger-Analyse ist eine Programmanalyse über allen Anweisungen und Ausdrücken von $\mathcal{P}$, die Referenzen verwenden. Der zu Grunde liegende Datentyp ist also $T = \{\text{alle Referenzen}\}$.

[3] In Abschnitt 4.3.4 hatten wir darauf hingewiesen, dass innerhalb unserer Programmrepräsentation zwischen Zeigern und Referenzen unterschieden wird. Die Analyse, die wir in diesem Abschnitt vorstellen, bezieht sich auf die Werte, die wir in *Firm* als „Referenzen“ bezeichnet haben. Da diese Analyse jedoch im allgemeinen Sprachgebrauch als „Zeigeranalyse“ bezeichnet wird, folgen wir diesem Beispiel und verzichten auf die Bezeichnung „Referenzanalyse“, auch wenn sie konsistent mit den Begriffen der zu Grunde liegenden Programmrepräsentation wäre.

Den Effekt einer Anweisung *op* auf den Programmzustand zur Laufzeit beschreiben wir in einer Programmanalyse durch eine geeignete Transferfunktion $Pto(op) : \mathcal{S} \mapsto \mathcal{S}'$ auf eine Abstraktion $\mathcal{S} \subseteq \mathcal{V}$ des Programmzustandes. Die Anforderung an die Definition von Transferfunktionen ist, dass jede Transferfunktion *sicher* ist. Eine Transferfunktion einer Anweisung *op* ist genau dann sicher, wenn sie eine gegebene Abstraktion $\mathcal{S}$ eines Zustandes, der der Ausführung von *op* besteht, nur in eine Abstraktion $Pto(op) : \mathcal{S} \mapsto \mathcal{S}' \subseteq \mathcal{V}$ abbildet, die mindestens die Zustände abstrahiert, die durch die Ausführung von *op* existieren können, nachdem *op* in einem von $\mathcal{S}$ abstrahierten Zustand ausgeführt wurde.

4.4.1 Eigenschaften unserer Zeigeranalyse

Zielsetzung Wir entwickeln unsere Zeigeranalyse so, dass sie für unsere Zwecke geeignet ist. Wir wollen mit unserer Analyse Programme im Ganzen analysieren, und wir wollen die Ergebnisse der Analyse mit Blick auf das gesamte Programm verwenden können. Es ist uns nicht besonders wichtig, Informationen über Details des Programms zu erfahren (wie dies bei vielen Übersetzeroptimierungen der Fall ist). Hingegen legen wir viel Wert darauf, dass die Analyseinformationen über die gesamte Struktur des Programmes hinweg aussagekräftig bleiben.

Die charakteristischen Eigenschaften unserer Zeigeranalyse stellen wir im Folgenden im Detail vor. Vorab erklären wir, welche Ziele bei der konkreten Ausgestaltung der Eigenschaften verfolgt wurden.

Das Namenschema, das wir in dem kommenden Abschnitt beschreiben, wählen wir sehr fein, um die zur Laufzeit allozierten Objekte in der Analyse möglichst gut unterscheiden zu können. Zur Abstraktion verwenden wir die Idee des funktionalen Speichers [Ste95], da wir durch die geeignete Wahl der Traversierung des Programmes erreichen können, dass wir ständig nur eine einzige Version der Abstraktion unterhalten müssen. Wir gestalten unsere Zeigeranalyse kontextsensitiv und wählen die Kontexte so, dass wir vornehmlich Prozeduren in unterschiedlichen Aufrufkontexten analysieren können. Auf die Analyse von Anweisungen in unterschiedlichen intraprozeduralen Kontexten legen wir keinen Wert.

Namen und Namenschema Wir abstrahieren die möglichen Werte von Referenzen, die zur Laufzeit des Programms auftreten können, mit *abstrakten Namen* (kurz *Namen*). Die Vorschrift, nach der Namen erstellt werden, ist ein *Namensschema*. Jedes Namensschema muss folgende grundlegende Eigenschaft erfüllen:

Definition 4.8 (Mindestanforderung an ein Namensschema) *Seien* $n_1, n_2 \in \mathcal{N}$ *zwei Namen, und sei für einen Namen* $n \in \mathcal{N}$: $objs(n)$ *die Menge aller Objekte, die*

durch n repräsentiert werden. Dann muss stets gelten:

$$n_1 \neq n_2 \quad \Longrightarrow \quad objs(n_1) \cap objs(n_2) \stackrel{!}{=} \emptyset.$$

Damit ergibt sich für die Zeigeranalyse $\mathcal{V}_{ref} = 2^{\mathcal{N}}$. Wir geben in Tabelle 4.2 einige Beispiele für Namensschemata an. Da zur Laufzeit Objekte an *alloc*-Operationen neu entstehen, während die anderen Operationen mit existierenden Objekten arbeiten, reicht es, wenn wir in Tabelle 4.2 angeben, welche Namen für *alloc*-Operationen zu erstellen sind.

Wahl von $\mathcal{N}$	**Beschreibung**
$\mathcal{N}_0 = \{n\}$	Alle Objekte werden auf einen Namen abgebildet.
$\mathcal{N}_T = \{n_t \mid t \in T\}$	Für jeden Typ $t \in T$ existiert ein Name.
$\mathcal{N}_a = \{n_i \mid i \in I\}$	Für jede *alloc*-Operation wird ein Name erstellt.
$\mathcal{N}_\Delta = \{n_{i,\delta} \mid i \in I, \delta \in \Delta\}$	Für jede *alloc*-Operation wird für jeden Kontext, in dem sie ausgeführt werden kann, ein Name erstellt.

Tabelle 4.2: Beispiele für Namensschemata

Schema $\mathcal{N}_0$ in Tabelle 4.2 ist ein triviales Schema, dass in Programmanalysen zur Anwendung kommt, die Haldenoperationen überhaupt nicht berücksichtigen. Schema $\mathcal{N}_T$ liegt der schnellen Typanalyse [BS96] und der Typ-basierten Alias-Analyse [DMM98] zu Grunde. Schema $\mathcal{N}_a$ ist ein einfaches Schema für schnelle Zeiger-Analysen [Str91]. In [Tra00] kommt ein Schema zum Einsatz, das eine zur Laufzeit der Analyse bestimmte Teilmenge von $\mathcal{N}_\Delta$ nutzt.

Für die Analyse wählen wir als die Wertemenge von Referenzen die Potenzmenge über dem zu Grunde liegenden Namensschema, also $2^{\mathcal{N}_\Delta}$. Wir stellen fest, dass mit $v_\top = 2^{\mathcal{N}_\Delta}$ bereits ein Supremum und mit $v_\bot = \emptyset$ ein Infimum existiert, und dass weiterhin mit der Teilmengenrelation $\cdot \subseteq \cdot$ eine geeignete Teilordnung und mit der Vereinigung $\cdot \cup \cdot$ und dem Schnitt $\cdot \cap \cdot$ geeignete binäre Operationen für die Bildung von Infimum und Supremum existieren. Wir definieren also die Abstraktion für Referenzen als

$$\mathcal{V}_{ref} = (2^{\mathcal{N}_\Delta}, \subset, \cup, \cap).$$

Reihungen und Zeigerarithmetik Um Reihungen aus Referenzen zu analysieren, unterscheiden wir nicht zwischen den einzelnen Einträgen der Reihung. Für Quellsprachen, die auf Referenen arithmetische Operationen zulassen (wie zum Beispiel `C` oder `C++`) eignet sich unser Namensschema nicht direkt, da es keine Namen vorsieht, die diejenigen Objekte zusammenfassen können, auf die mit berechneten Referenzen zugegriffen werden kann.

Funktionaler Speicher Um den Zustand des Haldenspeichers selber zu abstrahieren, orientieren wir uns an dem Ansatz des funktionalen Speichers [Ste95]. Durch die Wahl der Auswertungsreihenfolge während der Analyse vermeiden wir, dass wir unterschiedliche Versionen des funktionalen Speichers vorhalten müssen. Gegeben einen Namen $n \in \mathcal{N}$ und einen Felddeskriptor für ein Feld f einer Klasse c liefert uns der funktionale Speicher eine Abbildung

$$M(\delta, n, f) \to \mathcal{V}_{ref}. \quad (4.1)$$

Kontextsensitive Analyse Mit Blick auf die Anforderungen an unsere Zeigeranalyse wählen wir die Kontexte wie folgt:

Wir unterscheiden zwei Pfade genau dann, wenn sie sich in einem interprozeduralen Anteil unterscheiden. Unterscheiden sich zwei Pfade nur durch intraprozedurale Anteile, so ordnen wir sie dem gleichen Kontext zu. Damit existieren für jede Anweisung einer Prozedur p die gleichen Kontexte, nämlich die Menge aller Aufrufpfade, die zu p führen.

Prozeduren und Kontexte Wir können also einen Aufrufpfad über die Prozeduren $p_1, p_2, \ldots, p_{n-1}, p_n$ mit einem Kontext von p_n identifizieren:

$$\delta = (p_1, p_2, \ldots, p_{n-1}, p_n).$$

Um einen Kontext $\delta = (p_1, p_2, \ldots, p_{n-1})$ einer Prozedur p_{n-1} um einen Aufruf an p_n erweitern zu können, verwenden wir die Schreibweise

$$\delta' = (\delta, p_n) := (p_1, p_2, \ldots, p_{n-1}, p_n).$$

4.4.2 Transferfunktionen

Die Analyse der Operationen besteht aus der geeigneten Abbildung der konkreten Semantik der Operationen zur Laufzeit auf Transferfunktionen, die wir während der Analyse ausführen können:

Initialzustand: Den Zustand zu Beginn eines Programmlaufs modellieren wir mit einem Wert des abstrakten Speichers, der an allen Stellen (für alle möglichen Felder f) undefiniert ist:

$$Pto(\delta, \texttt{start}_{\texttt{main}}) = M, \text{ wobei}$$
$$M(\delta, n, f) = \bot \text{ für alle } n.$$

Prozedur: Wir analysieren eine Prozedur, indem wir ihre Anweisungen analysieren. Voraussetzung ist ein Speicherzustand M zu Beginn der Prozedur, und eine Belegung der formalen Parameter mit aktuellen Werten v_i, $i \leftarrow [1, \ldots, n]$. Das Ergebnis dieser Analyse ist ein Tupel, bestehend aus dem Speicherzustand M' am Ende der Prozedur und, wenn vorhanden, dem Rückgabewert r. Die Analyse einer Prozedur wird in einem vorgegebenen Kontext δ durchgeführt.

$$Pto(\delta, p, M, v_1, \ldots, v_n) = (M, r).$$

Feldauswahl: Die Auswahl von Objektfeldern und -methoden durch die *select*-Operation ändert den Speicherzustand nicht. Deshalb brauchen wir sie nicht eigens zu berücksichtigen. Wenn das Ergebnis der *select*-Operation in *load*-, *store*- oder *call*-Operationen einfließt, so errechnen wir für diese Operation den Deskriptor von dem Feld oder Methode.

Auswahl von Elementen einer Reihung: Wir unterscheiden in unserer Analyse nicht zwischen den einzelnen Elementen einer Reihung, sondern behandeln sie wie ein einzelnes Objekt.

Allokation: Das Analyseergebnis einer Allokationsoperation ist immer der Name, den wir für diese Operation im aktuellen Kontext vergeben haben. Für die Felder, die für die allozierte Klasse definiert sind, hat der resultierende Speicher zunächst keine Zuordnung. Entsprechend der Vorschrift der Quellsprache, dass alle referenzwertigen Felder eines Objektes nach der Allokation den `null`-Zeiger enthalten, wählen wir die Zuordnung des Funktionalen Speichers für das neu allozierte Objekt. Durch die Analyse von *store*-Operationen wird diese Zuordnung im Laufe der Analyse erweitert.

$$Pto(\delta, alloc(M)) = (\{n_{alloc,\delta}\}, M'), \text{ mit}$$
$$M'(\delta, n_{alloc,\delta}, f) = \bot \text{ für alle } f.$$

Feld laden: Für die Referenz r, über die der Lesezugriff durchgeführt wird, haben wir analysiert, welche Namen $n \in N_r = Pto((\delta, r))$ auftreten können. Um den Effekt einer *load*-Operation zu analysieren, werten wir den funktionalen Speicher an diesen Namen $n \in N_r$ aus. Die so erhaltenen Werte kombinieren wir wieder. Der Inhalt des funktionalen Speichers wird nicht geändert. Der Deskriptor des geladenen Feldes wird aus dem p-Operanden berechnet; er sei als f bekannt:

$$Pto(\delta, load(M, p)) = (M', p'), \text{ wobei}$$
$$M' = M$$
$$p' = \bigcup_{n \in Pto(\delta,p)} M(\delta, n, f)$$

Feld beschreiben: Für die Referenz r, über die der Schreibzugriff durchgeführt wird, haben wir analysiert, welche Namen $n \in N_r = Pto((\delta, r))$ auftreten können. Für die *store*-Operation simulieren wir für jeden dieser Namen den Effekt der Operation. Wir berechnen dazu einen neuen Zustand des Speichers, der diese Effekte berücksichtigt. Der Deskriptor des Feldes sei wieder f:

$$Pto(\delta, store(M, p, v, f)) = M' \text{ mit}$$
$$M' = M|\{[(\delta, n, f) \to V]| n \in Pto(p)\}, \text{ wobei}$$
$$V = \bigcup_{n' \in Pto(\delta, v)} M(\delta, n, f).$$

Zusammenfluss von Werten: Die Analyse einer Φ-Operation mit n Argumenten ist die Kombination der ankommenden Referenzen:

$$Pto(\delta, \Phi(p_1, \dots p_n)) = \bigcup_{i \leftarrow [1, \dots, n]} Pto(\delta, p_i).$$

Methodenaufruf: Um einen Methodenaufruf an eine Methode m von Klasse c zu analysieren, berücksichtigen wir alle möglichen Prozeduren $p_i \in impl^*(c, m)$, die als Implementation der aufgerufenen Methode in Frage kommen. Für jede Prozedur hinterlegen wir sowohl den aktuellen Zustand des funktionalen Speichers als auch die aktuellen Parameter, die wir beim Aufruf finden, an der Stelle der formalen Parameter der Prozedur. Werden als Argumente Zeiger übergeben, so übertragen wir aus der Feldauswahl des Aufrufers das dort ausgewählte Feld. Um das Ergebnis des Aufrufs zu erhalten, kombinieren wir die unterschiedlichen Speicherzustände, die von den aufgerufenen Prozeduren hinterlassen werden, und, sofern vorhanden, die Rückgabewerte der Prozeduren. Die Speicherzustände kombinieren wir punktweise.

$$Pto(\delta', call(M, m, v_1, \dots, v_n)) = (M', v), \text{ wobei}$$
$$(M', v) = \bigcup_{p \in impl^*(c,m)} Pto(\delta, p, M, v_1, \dots, v_n) \text{ und}$$
$$\delta' = (\delta, p).$$

4.4.3 Konstruktion des Aufrufgraphen

Über den Ablauf *innerhalb* einer Prozedur gibt uns der Grundblockgraph Auskunft. Um Klarheit über den interprozeduralen Ablauf zu erhalten, berechnen wir aus den einzelenen Prozeduren des Programmes den Aufrufgraphen.

Der Aufrufgraph ist ein bipartiter, gerichteter Graph $A \subseteq \{(p \to (c, m)) \cup ((c, m) \to p) : p \in P, (c, m) \in M_c\}$, wobei $M_C := \{(c, m) : c \in C, m \text{ ist Methode in } C\}$.

Für eine Prozedur $p \in P$ entält A genau eine Kante $(p \to (c, m))$ für jede Methode m der Klasse c, die innerhalb von p aufgerufen wird. Zwischen den Methodenaufrufen in p ist durch den intraprozeduralen Ablauf eine Teilordnung $T_i(p)$ zwischen den Aufrufen gegeben. Für ein p sind alle Kanten $\{(p \to (c, m))\} \subseteq A$ geordnet. Diese Ordnung darf $T_i(p)$ nicht widersprechen. Für jede Kante in $\{(p \to (c, m))\} \subseteq A$ enthält A die Kanten $\{(c, m) \to p' : p' \in impl^*(c, m)\}$.

4.4.4 Steuerung der Analysereihenfolge

Start der Analyse Wir beginnen die Analyse, indem wir die Programmeinsprungsprozedur des *Firm*-Programmes analysieren. Sofern diese Prozedur formale Parameter hat, deren Werte erst beim Aufruf des Programmes durch das Betriebssystem eingesetzt werden, müssen wir diese Parameter mit einem besonderen abstrakten Wert belegen. Dieser Wert ist ein Platzhalter für die noch unbekannten Werte, die zur Laufzeit auftreten können. In der Zeigeranalyse betrachten wir nur Objekte und Referenzen auf diese Objekte; da es nicht möglich ist, Objekte vor Beginn des Programmes zu allozieren, können sie auch nicht als Argumente der Programmeinsprungsprozedur vorkommen, so dass wir in der Implementierung keinen gesonderten abstrakten Wert vorhalten müssen.

Analyse einer Prozedur Wir analysieren die Operationen einer Prozedur in der Reihenfolge, in der sie sich auf den Speicherzustand auswirken. Dadurch vermeiden wir, dass wir gleichzeitig unterschiedliche Zustände des Hauptspeichers unterhalten müssen. Wie wir in Abschnitt 4.3.4 gesehen haben, ist in *Firm* für die Operationen, die sich auf den Speicherzustand auswirken, deren Abhängigkeit explizit über Kanten vom Modus M ausgedrückt. Um die Operationen in der gewünschten Reihenfolge zu besuchen, führen wir eine Postfix-Tiefensuche über die Kanten vom Modus M aus.

Analyse von Schleifen und Rekursionen Durch Anweisungen zur Ablaufsteuerung in der Quellsprache, durch die Schleifen ausgedrückt werden, und durch direkte oder indirekte rekursive Aufrufe zwischen Prozeduren erhalten wir Zyklen im *Firm*-Graphen. Zyklen durch Schleifen sind auf einen Graphen beschränkt, während durch Rekursionen interprozedurale Zyklen verursacht werden, die Kanten des Aufrufgraphen enthalten. In beiden Fällen bestimmen wir mit Hilfe einer Intervallanalyse eine konservative Abschätzung der Schleifen, und iterieren über diese Intervalle, bis die Analyseinformationen konvergieren.

Interprozedurale Analyse In Abschnitt 4.4.1 haben wir die Analyse eines Methodenaufrufs dadurch definiert, dass für alle möglichen aufgerufenen Prozeduren die aktuellen Analyseergebnisse zu Beginn der Ausführung an diese Prozeduren übergeben werden. Um innerhalb einer Prozedur p einen Methodenaufruf zu analysieren, unterbrechen wir die Analyse von p, iterieren über die aufgerufenen Prozeduren, um dann wieder zur Analyse von p zurückzukehren.

Nutzung der Typsicherheit Die Quellsprache *Java*, für die wir unsere Zeigeranalyse verwenden, ist dynamisch typsicher: Die Ausführung von Operationen auf Werten, deren Typen nicht kompatibel sind mit denen, die die Operation erwartet, wird zur Laufzeit unterbunden. Wir können deshalb an allen Stellen, an denen abstrakte Werte in Transferfunktionen eingehen, diejenigen Werte eliminieren, deren Typ mit dem erwarteten Typ der Transferfunktion nicht übereinstimmt. Dadurch erreichen wir durch lokale Massnahmen eine Erhöhung der Präzision der Analyse.

4.5 Bestimmung der Lebenszeiten von Objekten

4.5.1 Zugriffsanalyse

Wir haben bereits in Definition 4.2 beschrieben, wie ein Programm mit Haldenobjekten umgehen muss. In diesem Abschnitt widmen wir uns der Frage, wie wir die Zeigeranalyse einsetzen können, um diese Korrektheit zu erreichen. Wir lokalisieren die Zugriffe auf Objekte durch Inspektion der Operationen, die mit Referenzen arbeiten. Dabei unterscheiden wir zwischen den Operationen, zu deren Ausführung es erforderlich ist, dass ein Objekt tatsächlich alloziert ist, und den Operationen, die lediglich mit Referenzen arbeiten. Die Objekte, auf die zugegriffen wird, notieren wir an den entsprechenden Operationen.

Objektzugriffe Für eine Ladeoperation $load(M, r, f) \to (M', v)$ notieren wir die Namen $Pto(\delta, r)$. Analog notieren wir $Pto(\delta, r)$ für ein $store(M, r, f, v) \to M'$. Für ein $alloc(M) \to (M', r)$ notieren wir mit $Pto(\delta, r)$ die Namen, die in den unterschiedlichen Kontexten vergeben wurden. Die Auswahl von Methodenimplementierungen durch eine *select*-Operation müssen wir dann als Objektzugriff notieren, wenn im *Backend* diese Auswahl über Zuordnungstabellen implementiert wird. Bei der Verwendung von *fat pointers* entfällt dies. Ein direkter oder indirekter Aufruf an eine Prozedur, die Zugriffe auf den Namen enthält, notieren wir ebenfalls als Zugriff. Weitere Operationen, die auf Objekte zugreifen, existieren in *Firm* nicht und werden auch nicht durch spätere Phasen des Übersetzers eingefügt.

Operationen mit Referenzen In den verbleibenden Operationen werden zwar Referenzen verwendet, aber es ist nicht notwendig, dass zur Ausführung dieser Operationen das referenzierte Objekt tatsächlich auf der Halde alloziert ist. Zum Beispiel muss zur Ausführung von einem $store(M, r, v, f) \to M'$ oder einem $load(M, r, f) \to (M', v)$ nur das von r referenzierte Objekt alloziert sein. Ist v tatsächlich eine Referenz, die in dem Feld f gespeichert wird, so ist es für die o.g. Operationen nicht nötig, dass das durch v referenzierte Objekt auf der Halde alloziert ist. Gleiches gilt bei anderen Operationen mit Referenzen innerhalb von *Firm*, wie dem vergleich von Referenzen oder der Übergabe von Referenzen in eine Prozedur bzw. der Rückgabe einer Referenzen am Ende einer Prozedurausführung. Auch für Operationen, die erst durch spätere Phasen des Übersetzers erstellt werden, wie das Kopieren von Referenzen zwischen Prozessorregistern und Aufrufschachteln ist es nicht nötig, dass die referenzierten Objekte auf der Halde alloziert werden.

In diesen Fällen ist also die Referenz auf ein Objekt im Zustand des Programmes — also auf dem Laufzeitkeller oder in einem Feld eines Objektes auf der Halde — vorhanden. Für einen Speicherbereiniger würde es damit als erreichbar erscheinen, und sein Speicher könnte nicht freigegeben werden. Da wir aber mit der Zeigeranalyse die Semantik des Programmes analysieren, können wir an diesen Stellen vermeiden, dass die Präsenz einer Referenz gedeutet wird als ein Grund, das referenzierte Objekt alloziert zu lassen.

4.5.2 Analysekontexte

Um den nächsten Abschnitt vorzubereiten, entwickeln wir ein besseres Verständnis von Aufrufkontexten. In Abschnitt 4.4 hatten wir bereits Kontexte als Teilmengen von Pfaden zu einem Programmpunkt kennen gelernt, und wir hatten festgehalten, dass wir bei unserer Zeigeranalyse nur inerprozedurale Pfade unterscheiden.

Definition 4.9 (Aufgerufene Prozeduren, Aktivierung von Prozeduren) *Für eine Prozedur $p \in P$ seien mit $called(p) \subseteq P$ alle Prozeduren gemeint, die von p aus direkt oder indirekt aufgerufen werden.*

Solange eine Prozedur $p \in P$ oder eine Prozedur $p' \in called(p)$ ausgeführt wird, existiert eine Aktivierung *von p, das heisst, dass die Ausführung von p begonnen, jedoch noch nicht beendet wurde. Wir sagen auch, dass p* aktiv *ist.*

Für einen Kontext $\delta = (main, p_1, p_2, \ldots, p_{k-1}, p_k)$ sagen wir, dass sich das Programm $\mathcal{P}$ in Kontext δ befindet, wenn Prozedur p_n ausgeführt wird oder aktiv ist, während die Prozeduren $main, p_1, p_2, \ldots, p_{k-1}$ aktiv sind. Befindet $\mathcal{P}$ sich in δ, und wird dabei ein $p \in called(p_n)$ ausgeführt, so gibt es einen Kontext $\delta' = (main, p_1, p_2, \ldots, p_k, \ldots, p)$, mit dem wir den Kontext, in dem $\mathcal{P}$ ausgeführt

wird, genauer angeben können. Dann ist δ eine Obermenge aller Ausführungspunkte von δ'. Dies halten wir in der nächsten Begriffsbestimmung fest:

Definition 4.10 (Gemeinsame Präfixe von Kontexten) *Gegeben zwei Kontexte δ_1 und δ_2, so bezeichnen wir mit*

$$\overline{\delta} = \delta_1 \sqcap \delta_2$$

den längsten gemeinsamen Präfix von δ_1 und δ_2. Wir sagen

$$\overline{\delta} = \prod_{i \leftarrow [1 \ldots k]} \delta_i :\Longleftrightarrow \forall i \leftarrow [1 \ldots k] : \overline{\delta} \sqcap \delta_i = \overline{\delta}.$$

Ausserdem legen $\delta_1 \sqsubset \delta_2$ so wir fest, dass

$$\delta_1 \sqsubset \delta_2 :\Longleftrightarrow \delta_1 = \delta_1 \sqcap \delta_2.$$

Mit dieser Notation können wir nun folgendes feststellen: Für alle $p' \in called(p)$ finden wir, dass $\forall \delta' \in \Delta(p') : \exists \delta \in \Delta(p) : \delta \sqsubset \delta'$. Für alle $p' \in called(p)$, die ausgeführt werden, solange p aktiv ist, gilt: Wenn p im Kontext $\delta \in \Delta(p)$ ausgeführt wird, werden alle p' in einem Kontext δ' ausgeführt, für den $\delta \sqsubset \delta'$ gilt.

Auch umgekehrt können wir dies ausnutzen: Gegeben eine Menge von Prozeduren $P' \subseteq P$, $P' = \{p_1, p_2, \ldots, p_k\}$ und eine Menge Kontexte $\Delta' = \{\delta_1, \delta_2, \ldots, \delta_k\}$ für diese Prozeduren, so dass $\delta_i \in \Delta(p_i)$, können wir mit

$$\overline{\delta} := \prod_{i \leftarrow [1 \ldots k]} \delta_i$$

einen Kontext angeben, der die Obermenge aller Programmpunkte angibt, die alle $\delta_i \in \Delta'$ umfasst; jedes $\overline{\delta}' \sqsubset \overline{\delta}$ erfüllt diese Bedingung, und dabei ist $\overline{\delta}$ die *präziseste* Angabe einer Obermenge, die diese Bedingung erfüllt.

4.5.3 Lebenszeiten von Objekten

Mit den Kontexten aus dem vorigen Abschnitt können wir Fragmente des Programms angeben, die zur Laufzeit immer ein zusammenhängender Abschnitt der Programmausführung beschreiben. In diesem Abschnitt erklären wir, wie wir Kontexte so geeignet wählen können, dass sie die Lebenszeiten von Objekten bestmöglich umfassen.

Intraprozedurale Lebenszeiten Wenn in keinem der Aufrufer von einer Prozedur p auf Objekte zugegriffen wird, für die wir Zugriffe innerhalb von p notiert haben, können wir die Lebenszeit für diese Objekte innerhalb von p einfangen. Um die Zugriffe innerhalb von p einzufangen, betrachten wir alle Grundblöcke, innerhalb derer Zugriffe stattfinden, und bestimmen deren gemeinsamen Dominator und Postdominator. Aus der gängigen Literatur [WG84, WM96, Muc97, Mor98, Wir05] rufen wir uns die Definition in Erinnerung:

Definition 4.11 (Dominator und Postdominator) *Gegeben sei eine Prozedur $p \in P$ mit Startblock b_{start} und Endblock b_{end}. Weiterhin seien b und b' zwei Blöcke des Grundblockgraphen von p.*

Wir sagen b dominiert *b' gdw. jeder Pfad von b_{start} nach b' enthält b. Wir notieren dann: b dom b'.*

Wir sagen b' post-dominiert *b gdw. jeder Pfad von b nach b_{end} enthält b' und notieren b' pdom b.*

Für eine Menge von Blöcken B verallgemeinern wir dies und notieren:

$$\begin{aligned} b \; dom \; B' \quad &:\Leftrightarrow \quad \forall b' \in B' : b \; dom \; b' \text{ und} \\ B \; pdom \; b' \quad &:\Leftrightarrow \quad \forall b \in B : b \; pdom \; b'. \end{aligned}$$

Den Beginn der Lebenszeit der Objekte können wir mit dem Dominator und das Ende der Lebenszeit mit dem Postdominator beschreiben. Wir betrachten alle Grundblöcke $B_{acc}(n) := \{b \in BB(p) : b \text{ enthält einen Zugriff auf } n\}$ und bestimmen $\overline{b}$ und $\underline{b}$ so, dass

$$\begin{aligned} &\overline{b} \; dom \; B_{acc}(n) \text{ und} \\ &B_{acc}(n) \; pdom \; \underline{b}, \end{aligned}$$

und begrenzen die Lebenszeit von Objekten aus $objs(n)$ mit dem Anfang von $\overline{b}$ und dem Ende von $\underline{b}$.

Interprozedurale Lebenszeiten Existiert für eine Prozedur p ein Aufrufer p', so dass sowohl p als auch p' auf Objekte eines Namens n zugreifen, so enthält die Lebenszeit der Objekte $objs(n)$ die gesamte Laufzeit von p. Die Unterscheidung, an welcher Anweisung von p auf n zugegriffen wird, ist dann belanglos, und wir können einen einzigen Zugriff auf n in p betrachten.

Da eine Prozedur in mehr als in einem einzigen Kontext aufgerufen werden kann, und auf Namen zugreift, die in diesen unterschiedlichen Kontexten gültig sind, müssen wir die Zugriffe auf diese Namen getrennt nach Kontexten betrachten:

Für jeden Namen sammeln wir nun die Zugriffe, die innerhalb der Prozeduren des Programmes geschehen:

Definition 4.12 (Zugriffe auf einen Namen) *Gegeben sei ein Name* $n \in \mathcal{N}$*, auf den im Programm in den Prozeduren* $P' = \{p_i | i \leftarrow [1, \ldots, k]\} \subseteq P$ *zugegriffen wird. Dabei werde in Prozedur* p_i *auf* n *zugegriffen, wenn* p_i *im Kontext* $\delta_i \in \Delta(p_i)$ *ausgeführt wird. Dann sei:*

$$acc(n) := \{(p_i, \delta_i) | p_i \in P', \delta_i \in \Delta(p_i), i \leftarrow [1, \ldots, k]\}.$$

Weiterhin kürzen wir ab:

$$\Delta_{acc}(n) := \{\delta_i | (p_i, \delta_i) \in acc(n)\}$$

und

$$P_{acc}(n) := \{p_i | (p_i, \delta_i) \in acc(n)\}.$$

Mit dieser Notation können wir die interprozeduralen Lebenszeiten der Objekte beschreiben, die für einen Namen erstellt werden:

Definition 4.13 (Interprozedurale Lebenszeiten der Objekte eines Names) *Gegeben sei ein Name* n*, auf den in den Prozeduren* $P_{acc}(n)$ *zugegriffen wird. Weiterhin seien* $\Delta_{acc}(n)$ *die Kontexte, in denen diese Zugriffe auftreten. Wir setzen*

$$\delta_{acc}(n) := \prod_{i \leftarrow [1 \ldots k]} \Delta_{acc}(n) \tag{4.2}$$

und charakterisieren wir die interprozedurale Lebenszeit *der Objekte, die mit* n *zusammengefasst sind, folgendermaßen:*

1. *Ist* $\delta_{acc}(n) = (main, \ldots, p)$*, so begrenzen wir innerhalb von* p *die Lebenszeit der Objekte* $objs(n)$ *durch den jeweils gemeinsamen Dominator und Postdominator aller Zugriffe innerhalb von* p*. Methodenaufrufe, die direkt oder indirekt zu Prozeduren in* $P_{acc}(n)$ *führen, betrachten wir dabei selber als Zugriffe auf* n*.*

2. *Alle Prozeduren in* $P_{acc}(n) \backslash p$ *gehören zum Zugriffsbereich von* n*, wenn sie im Kontext* $\delta_{acc}(\delta)$ *aufgerufen werden.*

Damit haben wir den präzisesten Bereich angegeben, inerhalb dessen die Zugriffe auf die Objekte $objs(n)$ stattfinden. Wir stellen noch fest, dass jeder Kontext $\delta' \sqsubset \delta_{acc}(n)$ ebenfalls ein gültiger Bereich ist, der alle Zugriffe auf Objekte von $objs(n)$ umfasst, wenngleich er grösser als der durch $\delta_{acc}(n)$ beschriebene Bereich ist.

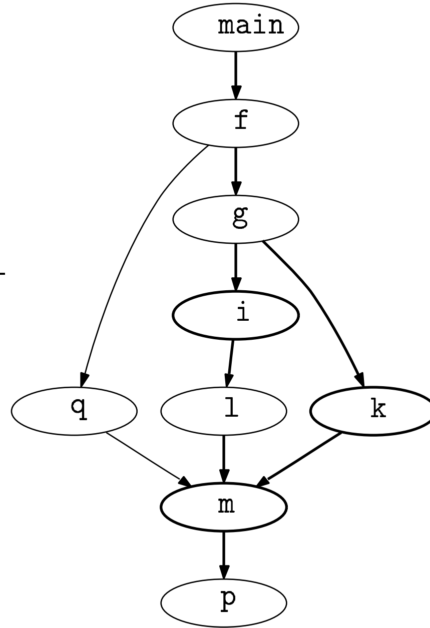


Abbildung 4.4: Aufrufgraph zu Beispiel 4.14

Beispiel 4.14 (Interprozedurale Zugriffsbereiche) *Wir betrachten ein Programm, dessen Aufrufgraph in Abbildung 4.4 angegeben ist. Wir bezeichnen*

$$\delta_g := (\mathtt{main}, \mathtt{f}, \mathtt{g}),\ \delta_i := (\delta_\mathtt{g}, \mathtt{i}),\ \delta_l := (\delta_i, \mathtt{l}), \delta_k := (\delta_i, \mathtt{k}) \textit{ und } \delta_p := (\delta_g, \mathtt{k}, \mathtt{m}, \mathtt{p}).$$

Die Zeigeranalyse vergibt für eine Allokationsanweisung in Prozedur $\mathtt{p}$ *im Kontext* $\delta_\mathtt{p}$ *den Namen* n*, und wir entnehmen ihren Ergebnissen:*

$$acc(n) = \{(\delta_\mathtt{i}, \mathtt{i}), (\delta_\mathtt{l}, \mathtt{l}), (\delta_\mathtt{k}, \mathtt{k}), (\delta_\mathtt{p}, \mathtt{p})\}.$$

Daraus berechnen wir nach Gleichung (4.2):

$$\delta_{acc}(n) = \bigsqcap\{\delta_\mathtt{i}, \delta_\mathtt{l}, \delta_\mathtt{k}, \delta_\mathtt{p}\} = \delta_\mathtt{g}.$$

Die Lebenszeit von Objekten, die mit dem Namen n *zusammengefasst werden, können wir also interprozedural durch die Laufzeit einer Instanz von Prozedur* $\mathtt{g}$ *abschätzen.*

Die Allokationsanweisung in Prozedur $\mathtt{p}$ *wird, wie in Abb. 4.4 ersichtlich, auch in den Kontexten* $\delta'_p = (\mathtt{main}, \mathtt{f}, \mathtt{g}, \mathtt{l}, \mathtt{m}, \mathtt{p})$ *und* $\delta''_p = (\mathtt{main}, \mathtt{f}, \mathtt{q}, \mathtt{m}, \mathtt{p})$ *ausgeführt. Wie wir diesen Fall behandeln, beschreiben wir später in Abschnitt 4.8.*

4.6 Allokation in Regionen

Nachdem wir in Abschnitt 4.5.3 festgestellt haben, wie wir die Lebenszeiten von Objekten einfangen können, widmen wir uns nun der Frage der praktischen Umsetzung. Wir kennen nun Programmpunkte, die Anfang und Ende eines zusammenhängenden Teils der Programmausführung darstellen, innerhalb dessen die Objekte erstellt werden, und innerhalb dessen auf die Objekte zugegriffen wird. Am Ende dieses Teils der Programmausführung müssen wir dafür sorgen, dass der Speicher, der für die Objekte alloziert wurde, wieder freigegeben wird.

Wir können nicht darauf bauen, alle Objekte durch Traversierung des vom Programm erstellten Programmgraphen zu erreichen, weil es möglich ist, dass einige der Objekte inzwischen nicht mehr erreichbar sind. Ausserdem würde sich die Frage stellen, ob in jedem Fall eine geeignete Traversierung des Objektgraphen gefunden werden kann, die genau die zu deallozierenden Objekte erreicht.

Um diese Probleme zu umgehen, allozieren wir die Objekte in einer Behälter-Datenstruktur. In Anlehnung an die Terminologie, die wir in Abschnitt 3.6 kennen gelernt haben, nennen wir sie *Regionen* [Bla99, CCQR04, CV98, GA01, GA98, Hal99, Han90, Kem95, NKY04, QH02, TB98, TB97, TT93, WSB01].

Beim Eintritt in einen Zugriffsbereich erstellen wir eine Region, die Allokationen legen die Objekte in der Region ab, und beim Verlassen des Zugriffsbereich verwerfen wir die Region wieder, und damit allen Speicher, der bis dahin für die Objekte alloziert worden ist.

In der Prozedur $p = \delta_{acc}(n)$, die wir mittels Gleichung (4.2) für einen Namen n ermittelt haben, erstellen und verwerfen wir eine Region ρ_n. Während der Ausführung aller $p' \in P_{acc}(n) \setminus \{p\}$ wird vorausgesetzt, dass ρ_n existiert; wir können uns — formell — für eine Prozedur $p' \in P_{acc}(n) \setminus \{p\}$ die Signatur um einen formellen Parameter ρ erweitert denken, der zur Ausführung von p in einem Kontext δ mit ρ_n besetzt wird. Die Regionen, die für eine Prozedur p angelegt sein müssen, bevor p in Kontext δ ausgeführt wird, nennen wir $\overline{\rho}_\delta(p)$.

Um den Deskriptor einer Region, den wir bei der Erstellung der Region erhalten, an die Allokationsanweisung zu übergeben, können wir die Signatur der Prozeduren um Parameter erweitern, oder wir können eine globale Variable generieren, in der der Deskriptor abgelegt wird.

4.7 Regionen

Implementierung Die Datenstruktur, die wir zur Implementierung unserer Regionen verwenden, ist als verkettete Liste von Datenblöcken ausgelegt. Die Grösse der Blöcke ist so gewählt, dass sie die durchschnittliche Grösse der Objekte um ein

Vielfaches übersteigt, so dass in einem Block der Region mehrere Objekte angelegt werden können.

Um über den Stand der Allokationen innerhalb der Blöcke Buch zu führen, ist eigentlich eine separate Kopf-Datenstruktur nötig, die

aber in den Anfang des ersten Datenblockes integriert ist. Steht innerhalb des aktuell letzten Datenblockes nicht genug Speicher zur Verfügung, um eine anstehende Allokation durchzuführen, so wird ein neuere Block von der Freispeicherverwaltung angefordert, in die Liste eingehängt, und das Objekt wird am Anfang des neuen Blockes alloziert.

Speicherverschnitt Wenn ein Objekt nicht in dem aktuell ersten Datenblock alloziert werden kann, weil dort nicht mehr genug Speicher verfügbar ist, bleibt dieser Speicher ungenutzt. Wir bezeichnen die minimale Ausrichtung des Speichers (in Bytes) mit a, und die Größe eines Datenblocks $d = d' \cdot a$ (ausschließlich des Verkettungszeigers). Wenn wir in einer Region Objekte allozieren, die jeweils $k = k' \cdot a$ Bytes benötigen, können pro Datenblock höchstens $v_o \leq (k' - 1) \cdot a$ Bytes Speicher ungenutzt bleiben. Allozieren wie n Objekte in der Region, so brauchen wir $n_d = \lceil n / \lfloor d'/k' \rfloor \rceil$ Datenblöcke. Dabei kann höchstens

$$v \leq v_o \cdot (n_d - 1) + d - (n - (n_d - 1) \cdot \lfloor b'/k' \rfloor) \cdot k \text{ Bytes}$$

Verschnitt entstehen.

Freispeicherverwaltung für Regionen Wenn alle Objekte des Programms in Regionen alloziert werden, so werden von der zu Grunde liegenden Freispeicherverwaltung nur noch Datenblöcke angefordert bzw. an sie zurückgegeben. Ihre Aufgabe reduziert sich damit auf die Verwaltung von Blöcken, die alle die gleiche Grösse haben. Problematische Effekte wie externer Verschnitt und Speicherfragmentierung entfallen damit.

Regionen-Polymorphie Wir haben nun die Lebenszeiten von Objekten bestimmt, und wir fangen die Objekte dadurch ein, dass wir sie in Regionen allozieren. Für jede Prozedur p existiert eine Menge von Regionen $\overline{\rho}(p) = \{\rho_i\}$, die angelegt sein müssen, wenn die Ausführung von p ansteht. Wir können uns — formell — die Signatur von einer Prozedur p um $\overline{\rho}(p)$ erweitert denken, um zu beschreiben, dass die Regionen von $\overline{\rho}(p)$, genau so wie die aktuellen Parameter von p, bei ihrem Aufruf zur Verfügung stehen müssen.

Die Situation, auf die wir hier gestossen sind, ist ähnlich der, die wir bereits bei den funktionalen Sprachen angetroffen haben, wo sie als *Regionen-Polymorphie* [TT93] bezeichnet wird. Im Gegensatz zu diesen Ansätzen haben wir

jedoch die Regionen $\overline{\rho}(p)$ für jeden Aufrufkontext einzeln berechnet, nicht für alle Kontexte gleichzeitig. In Abhängigkeit davon, in welchem Kontext $\delta \in \Delta(p)$ die Prozedur p nun aufgerufen wird, können sich die erweiterten Signaturen $\overline{\rho}_\delta(p)$ von p unterscheiden. Auf die Konflikte, die sich daraus ergeben können, gehen wir im nächsten Abschnitt ein.

4.8 Konflikte zwischen Regionen

4.8.1 Entstehen von Konflikten

Wir betrachten eine Allokationsanweisung *alloc*. Sie werde in einer Prozedur p in zwei Kontexten $\{\delta_1, \delta_2\} \subseteq \Delta(p)$ ausgeführt. In den beiden Kontexten werden für die Anweisung die Namen n_1 und n_2 vergeben. Geeignete Zugriffsbereiche bestimmen wir nach Gleichung (4.2) und finden die Kontexte $\delta_{acc}(n_i)$, $i \leftarrow [1,2]$. Es sei $\delta_{acc}(n_i) = (main, \ldots, p_i)$ für $i \leftarrow [1,2]$. Ist nun $p_2 \in called(p_1)$, dann ist $\delta_{acc}(n_2)$ vollständig in $\delta_{acc}(n_1)$ enthalten. Dies würde bedeuten, dass, wenn das Programm in Kontext $\delta_{acc}(n_1)$ ausgeführt wird, für den *alloc* mit einer Region ρ_1 parametrisiert wird, auch der Zugriffsbereich für $\delta_{acc}(n_2)$ betreten und verlassen wird. Nach dem in Abschnitt 4.6 gesagten würde dies bedeuten, dass dabei für $\delta_{acc}(n_2)$ ebenfalls eine Region ρ_2 angelegt und verworfen wird, dass also *alloc* mit dem Deskriptor von ρ_2 parametrisiert würde. Dann werden aber die Objekte von *alloc* in ρ_2 angelegt. Da jedoch $p_2 \in called(p_1)$ gilt, und damit $\delta_{acc}(n_2)$ eine kürzere Lebensdauer als $\delta_{acc}(n_1)$ hat, kann es passieren, dass ρ_2 verworfen wird, bevor der letzte Zugriff auf die Objekte von $objs(n_1)$ erfolgt ist; insbesondere kann *alloc* erneut ausgeführt werden, während es mit dem Deskriptor der inzwischen verworfenen Region ρ_2 parametrisiert wird. Wir geben geben ein Beispiel für einen Konflikt zwischen zwei Zugriffsbereichen an.

Beispiel 4.15 (Konflikte zwischen zwei Zugriffsbereichen) *Wir betrachten die Prozeduren main, e, f, g und k in Abbildung 4.5. Es ist* $\Delta(k) = \{\delta_1, \delta_2\}$ *mit*

$$\begin{aligned} \delta_1 &= (main, e, g, k) \text{ und} \\ \delta_2 &= (main, f, g, k). \end{aligned}$$

Weiterhin ist $\Delta(g) = \{\delta'_1 := (main, e, g), \delta'_2 := (main, f, g)\}$, $\Delta(e) = \{(main, e)\}$ *und* $\Delta(f) = \{\delta_f := (main, f)\}$. *Für die Ausführung von alloc in k seien die Namen* n_1 *in* δ_1 *und* n_2 *in* δ_2 *vergeben. Mit den Zugriffen auf* `obj` *in f, g und k ergibt sich*

$$\begin{aligned} acc(n_1) &= \{(g, \delta'_1), (k, \delta_1)\} \text{ und} \\ acc(n_2) &= \{(f, \delta_f), (g, \delta'_2), (k, \delta_2)\}. \end{aligned}$$

```
global obj;

main () {
  e();
  f();
}

e () {
  g();
}

f () {
  ρ ← newRegion();
  g();
  obj.f ← v;
  delete(ρ);
}

g () {
  ρ ← newRegion();
  k();
  obj.f ← v';
  delete(ρ);
}

k () {
  obj ← new@ρ;
}
```

Abbildung 4.5: Programmcode zu Beispiel 4.15

Wir bestimmen aus Gleichung (4.2)

$$\begin{aligned} \delta_{acc}(n_1) &= \delta'_1 = (main, e, g) \text{ und} \\ \delta_{acc}(n_2) &= \delta_f = (main, f). \end{aligned}$$

und vergeben die Region ρ_1 für n_1 in g und ρ_2 in f für n_2.

Da jedoch $g \in called(f)$ ist, ergibt sich die Programmausführung in Abbildung 4.6. In Zeile 16 wird nun der Paramater ρ, mit dem die Allokationsanweisung alloc parametrisiert wird, fälschlicherweise mit dem Deskriptor von ρ_2 überschrieben. In Zeile 18 wird das Objekt damit in ρ_1 angelegt. Diese Region jedoch wird in Zeile 21 verworfen, während in Zeile 23 noch auf das fälschlicherweise in ρ_1 angelegte Objekt zugegriffen wird. Erst in Zeile 24 wird ρ_2 verworfen, nachdem alle Zugriffe auf das Objekt erfolgt sind. Durch den Konflikt ist allerdings bereits $\rho = nil$ gesetzt worden, so dass diese Anweisung keinen Effekt hat.

4.8.2 Auflösen von Konflikten

Um mit den möglichen Konflikten zwischen Zugriffsbereichen ungehen zu können, haben wir zwei Möglichkeiten gefunden, die wir im Folgenden vorstellen. Wir beschreiben sie anhand der beiden Zugriffsbereiche $\delta_{acc}(n_1) = (main, \ldots, p_1)$

	Programmausführung	Zustand ρ	Zustand obj	Regionen
	`reference` obj;			
	`region` ρ;			
(1)	`begin main` ()	$\rho = nil$	$obj = nil$	
(2)	`begin` e()			
(3)	`begin` g()			
(4)	$\rho \leftarrow$ `newRegion` ()	$\rho = \rho_1$		$\rho_1 = \{\}$
(5)	`begin` k ()			
(6)	$obj \leftarrow$ `new@`ρ		$obj = o_1$	$\rho_1 = \{o_1\}$
(7)	`end` k()			
(8)	`obj.f` $\leftarrow v$		$obj = o_1$	
(9)	`delete` (ρ)	$\rho = \utilde{\rho_1}$		$\rho_1 = \perp$
(10)	`end` g()			
(11)	`end` e()			
(12)				
(13)	`begin` f ()			
(14)	$\rho \leftarrow$ `newRegion` ()	$\rho = \rho_2$		$\rho_2 = \{\}$
(15)	`begin` g()			
(16)	$\rho \leftarrow$ `newRegion` ()	$\rho = \rho_3$		$\rho_2 = \{\}$, $\rho_3 = \{\}$
(17)	`begin` k ()			
(18)	$obj \leftarrow$ `new@`ρ	$\rho = \rho_3$	$obj = o_2$	$\rho_2 = \{\}$, $\rho_3 = \{o_2\}$
(19)	`end` k()			
(20)	`obj.f` $\leftarrow v'$		$obj = o_2$	
(21)	`delete` (ρ)	$\rho = \utilde{\rho_3}$		$\rho_2 = \{\}$, $\rho_3 = \perp$
(22)	`end` g()			
(23)	`obj.f` $\leftarrow v''$		$obj = \utilde{o_2}$	
(24)	`delete` (ρ)	$\rho = \utilde{\rho_3}$		$\rho_2 = \{\}$, $\rho_3 = \perp$
(25)	`end` f()			
(26)	`end main` ()			

Abbildung 4.6: Programmausführung zu Beispiel 4.15. Deskriptoren und Referenzen sind dort, wo sie ungültig sind, durch unterstrichene Symbole ($\utilde{abc}$) markiert.

und $\delta_{acc}(n_2) = (main, \dots, p_2)$, die für die Namen n_1 bzw. n_2 einer Allokationsanweisung *alloc* in Prozedur p_a für die Kontexte δ_1 und δ_2 erstellt wurden. Der Konflikt bestehe darin, dass $p_2 \in called(p_1)$ ist.

Aufweiten der Zugriffsbereiche Wir hatten in Abschnitt 4.5.2 bereits festgestellt, dass wir mit Gleichung (4.2) immer die bestmögliche Näherung für einen Zugriffsbereich berechnen, und dass für einen Zugriffsbereich $\delta_{acc}(n)$ auch jeder Bereich $\delta \sqsubset \delta_{acc}(n)$ ein gültiger, wenngleich gröberer Zugriffsbereich ist. Solange nun Konflikte zwischen zwei Zugriffsbereichen bestehen, können wir einen der beiden Bereiche so aufweiten, dass keine Konflikte mehr existieren. Dazu unterscheiden wir zwei Fälle:

1. Es ist $\delta_{acc}(n_1) \sqsubset \delta_{acc}(n_2)$. Für alle $\delta = (main, \dots, p) \sqsubset \delta_{acc}(n_2)$ gilt nun einer der folgenden drei Fälle:

   - Es ist $\delta_{acc}(n_1) \sqsubset \delta$ und $\delta_{acc}(n_1) \neq \delta$. Damit ist immer noch $p \in called(p_1)$ und δ steht mit $\delta_{acc}(n_1)$ in Konflikt.
   - Es ist $\delta \sqsubset \delta_{acc}(n_1)$ und $\delta_{acc}(n_1) \neq \delta$. Nun steht $\delta_{acc}(n_1)$ umgekehrt mit δ in Konflikt.
   - Es ist $\delta = \delta_{acc}(n_1)$, und ein Konflikt tritt nicht auf.

   In diesem Fall können wir also den Konflikt beseitigen, indem wir den Zugriffsbereich von n_2 auf $\delta_{acc}(n_1)$ aufweiten.

2. Es ist *nicht* $\delta_{acc}(n_1) \sqsubset \delta_{acc}(n_2)$. Wir bezeichnen $\overline{\delta_{acc}(n_2)} := \delta_{acc}(n_1) \sqcap \delta_{acc}(n_2)$ und betrachten $\Delta_{n_2} := \{\delta = (main, \dots, p) : \overline{\delta_{acc}(n_2)} \sqsubset \delta \sqsubset \delta_{acc}(n_2), p \notin called(p_1)\}$.

   Ist nun $\Delta_n \neq \emptyset$, so gilt: Durch die Konstruktion von Δ_{n_2} sind alle $\delta \in \Delta_{n_2}$ gültige Zugriffsbereiche für n_2, die mit $\delta_{acc}(n_2)$ *nicht* in Konflikt stehen. Aus Δ_{n_2} wählen wir den längsten Eintrag $\overline{\delta}$ aus, und ersetzen $\delta_{acc}(n_2)$ durch $\overline{\delta}$.

   Ist jedoch $\Delta_n \stackrel{!}{=} \emptyset$, so verfahren wir wie in Fall 1, und legen $\delta_{acc}(n_2)$ mit $\delta_{acc}(n_1)$ zusammen.

Referenzzähler Um zu verhindern, dass beim Betreten und Verlassen von δ_2 eine Region angelegt bzw. verworfen wird, obwohl wir uns in δ_1 befinden, können wir auch einen Mechanismus einsetzen, der zur Laufzeit arbeitet. Wenn sich das Programm im Zugriffsbereich von $\delta_{acc}(n_1)$ aufhält, und den Zugriffsbereich von $\delta_{acc}(n_2)$ betritt, muss erkannt werden, dass bereits eine Region angelegt ist; wenn der Bereich von $\delta_{acc}(n_2)$ verlassen wird, muss verhindert werden, dass die

Region verworfen wird, obwohl sie erst am Ende von $\delta_{acc}(n_1)$ verworfen werden kann. Dies können wir dadurch erreichen, dass wir die Regionen mit einem Referenzzähler `rc` ausstatten, der wie folgt verwendet wird:

Programmstelle	**Aktion**

Eintritt in $\delta_{acc}(n)$	*if* $(\rho \neq nil)$ {
	 $\rho = \texttt{newRegion}()$;
	 } *else* {
	 $\rho \rightarrow rc{+}{+}$;
	 }

Austritt aus $\delta_{acc}(n)$	$\rho \rightarrow rc{-}{-}$;
	 if $(0 \stackrel{!}{=} \rho \rightarrow rc)$ {
	 $\texttt{deleteRegion}(\rho)$;
	 $\rho = nil$;
	 }

Mit dieser Technik können wir die Zugriffsbereiche so einsetzen, wie in Gleichung (4.2) errechnet. Die Aufweitung und die damit einhergehende Vergröberung der Zugriffsbereiche kann unterbleiben. Im Gegenzug verursachen wir die Kosten, die bei der Berechnung der Änderungen der Referenzzähler entstehen.

4.9 Zusammenfassung

Ansatz Wir haben am Anfang dieses Kaptitels in Abschnitt 4.1 die Ziele für unsere Arbeit gesetzt. In Abschnitt 4.3 haben wir vorgestellt, auf welcher Programmrepräsentation wir unsere Arbeit durchführen. In Abschnitt 4.3.2 haben wir erklärt, wie wir die Anweisungen einzelner Prozeduren darstellen, und in Abschnitt 4.3.3 haben wir die Darstellung der Datenstrukturen, und in Abschnitt 4.3.5 haben wir die Eigenschaften und Mechanismen vorgestellt, die zur Darstellung objektorientierter Programme nötig sind.

Korrektheit In Abschnitt 4.2 haben wir uns darüber Klarheit verschafft, welche Fehler beim Umgang mit Objekten auf der Halde vermieden werden müssen, und welche grundlegenden Eigenschaften für den Zugriffe auf Haldenobjekte gelten müssen, damit ein Programm keine Speicherfehler verursacht.

Analyse und Transformation In Abschnitt 4.4 haben wir unsere Implementierung der Zeiger-Analyse und in Abschnitt 4.5.1 das Konzept der Zugriffsanalyse vorgestellt. In Abschnitt 4.5.2 haben wir den Begriff des Analyse-Kontexts genutzt, um für auf der Halde allozierte Objekte geeignete Zugriffsbereiche zu definieren. In

Abschnitt 4.5.3 haben wir beschrieben, wie wir die Objekte eines Zugriffsbereiches einfangen, und in Abschnitt 4.8, wie wir Konflikte zwischen den Zugriffsbereichen finden und beseitigen können.

Im kommenden Abschnitt werden wir darlegen, welches Verhalten unser Ansatz in der Praxis zeigt.

Kapitel 5

Ergebnisse

5.1 Einführung

Nachdem wir in den vorangegangenen Kapiteln unsere Methode vorgestellt haben, stellen wir in diesem Kapitel vor, welche Resultate wir mit unserer Methode erzielen.

Zunächst widmen wir uns der Wahl geeigneter Testprogramme und beschreiben Details der Implementierung und der Durchführung der Testläufe. Wir zeigen Laufzeitergebnisse und analysieren das Laufzeitverhalten der Testprogramme. Abschließend beurteilen wir die vorgestellten Ergebnisse.

5.2 Auswahl der Benchmarks

An die Benchmark-Programme stellen wir den Anspruch, dass die Operationen, die wir mit unserem Ansatz ändern — Allokationen und Haldenoperationen — in ausreichender Häufigkeit ausgeführt werden. Innerhalb der SPEC-Suite finden sich jedoch keine Benchmarks, die dieses Kriterium auch nur annähend erfüllen. Wir greifen deshalb auf die *Java*-Portierung der Olden-Benchmarks zurück.

Beschreibung der Olden-Benchmarks In den Olden-Benchmarks sind folgende Programme enthalten:

Barnes-Hut (`BH`): Löst das *N*-Körper-Problem mit einem hierarchischen Ansatz.

Bitonic Sort (`BiSort`): Sortiert eine Menge von Objekten, indem zwei disjunkte bitonische Sequenzen verschmolzen werden.

Elektromagnetische Wellen, 3D (`Em3d`): Simuliert die Propagation elektromagnetischer Wellen innerhalb eines 3d-Objekts.

Health System Simulation (`Health`): Simuliert das kolumbianische Gesundheitssystem.

Minimal Spanning Tree (`MST`): Berechnet den minimalen spannenden Baum eines Graphen.

Perimeter (`Perimeter`): Berechnet den Durchmesser einer Menge von *Raster-Images*, die durch Quad-Bäumen kodiert werden.

Power Grid Pricing (`Power`): Löst das in [LML+93] beschriebene *Power Problem*.

Tree Addition (`TreeAdd`): Summiert Werte in einem Baum.

Travelling Salesman Problem (`TSP`): Berechnet eine Näherung für den besten hamiltonschen Pfad eines Handlungsreisenden-Problems.

Voronoi-Diagramm (`Voronoi`): Berechnet das Voronoi-Diagramm einer Menge von Punkten.

Umfang der Olden-Benchmarks Wir geben in Tabelle 5.1 den Umfang der Benchmarks in Zeilen Code an. Dabei werden leere Zeilen und Kommentarzeilen mitgezählt. Nicht mitgezählt wird der Umfang der Laufzeitumgebung, die den Programmen eine grundlegende Infrastruktur zur Verfügung stellt. An den Werten

Benchmark	Zeilen Code
`BH`	1277
`BiSort`	407
`Em3d`	607
`Health`	692
`MST`	563
`Perimeter`	877
`Power`	805
`TreeAdd`	267
`TSP`	620
`Voronoi`	1342

Tabelle 5.1: Umfang der Benchmarks

von Tabelle 5.1 erkennt man, dass es sich bei diesen Benchmarks um sog. *Kernel*-Benchmarks handelt, die also einen einzigen, bestimmten Algorithmus implementieren. Wir haben es bei diesen Benchmarks also mit Komponenten zu tun, die auch als ein Modul von grösseren Systemen vorkommen könnten. Diese Wahl ermöglicht es uns deshalb, das Verhalten unseres Ansatzes auf überschaubaren, klar

definierten Algorithmen zu untersuchen, und zu beobachten, wie der von diesen Komponenten benötigte Speicher genutzt wird, solange die jeweilige Komponente aktiv genutzt wird. Aufgrund unserer Implementation der Zeigeranalyse, die wir in Abschnitt 4.4 beschrieben haben, ist es uns leider nicht möglich, grosse Systeme im Ganzen zu analysieren, da die Laufzeit unserer Analyse exponentiell mit der Anzahl der im Programm sich ergebenen Kontexte steigt. Um grössere Softwaresysteme im Ganzen zu analysieren, kann man auf Zeigeranalysen wie die von Hirzel, Diwan und Hind [HDH04] oder die Ideen von Trapp [Tra00] zurückgreifen, deren Ergebnisse uns in dieser Arbeit leider nicht in einer verwertbaren Form zur Verfügung standen.

Während wir also unseren Ansatz aus technischen Gründen nicht an grossen Systemen im Ganzen demonstrieren können, können wir doch zeigen, wie sich unser Ansatz bei Teilsystemen verhält, die als Teil eines grossen Systems einen Dienst innerhalb eines grossen Systems liefern. Da, wo die Benchmarks in unseren Versuchen abgelaufen sind, würden sie das Ergebnis dieses Dienstes an ihren Aufrufer zurückliefern. Der Haldenspeicher, der zur Ausführung des Dienstes lokal alloziert wurde, und der nicht selber das Ergebnis ausmacht, kann dann dealloziert werden, während diejenigen Objekte, die das Ergebnis selber darstellen, an den Aufrufer zurückgegeben werden, der — nachdem er diese Daten ja angefordert hat — mit hoher Wahrscheinlichkeit auf diese Datenstruktruen zugreifen wird.

5.3 Quantifizierung der Analyse- und Transformationsergebnisse

Nach der Übersetzung der Benchmarks haben wir die Anzahl von Allokationsanweisungen und die Anzahl an Regionenanweisungen (jeweils Paare „Region erstellen“ und „Region verwerfen“) analysiert. Die Ergebnisse sind in Tabelle 5.2 aufgeführt. Mit unserem Ansatz setzen wir in das Programm neue Anweisungen ein, die Regionen anlegen und verwerfen. Dadurch erhöht sich der Umfang des Programmes. die Erhöhung des Umfanges des Programmes kann sich zum einen negativ auf die Laufzeit auswirken, da mehr Code geladen und ausgeführt werden muss. Zum andern kann sie dazu führen, dass der Speicher, der für den Code des ursprünglichen Programmes noch gross genug war, zu klein ist, um das transformierte Programm aufzunehmen.

Da die Regionen immer für existierende Allokationsanweisungen eingesetzt werden, führen wir in Tabelle 5.2 zusätzlich noch das Verhältnis zwischen Allokations- und Regionenanweisungen auf.

Für jede Allokationsanweisung muss mindestens eine Region angelegt und verworfen werden, so dass das Verhältnis immer ≥ 1 sein muss. An den Werten erkennt

Benchmark	Absolut		Verhältnis
	Allokationen	Regionen	Regionen zu Allokationen
BH	36	39	1.08
BiSort	4	7	1.75
Em3d	17	22	1.29
Health	23	26	1.13
MST	13	16	1.23
Perimeter	11	14	1.27
Power	25	28	1.12
TreeAdd	6	9	1.50
TSP	4	7	1.75
Voronoi	31	34	1.10

Tabelle 5.2: Allokations- und Regionenanweisungen

man, dass es regelmäßig nur wenig mehr Regionen- als Allokationsanweisungen existieren. Daraus lässt sich schließen, dass die Allokationsanweisungen entweder nur in wenigen unterschiedlichen Kontexten ausgeführt werden, oder dass die Zugriffsbereiche, die sich für die unterschiedlichen Kontexte ergeben, häufig in Konflikt stehen wie am Ende von Abschnitt 4.8 beschrieben, und dass sie deshalb zusammengefasst werden. Da es sich bei den Benchmarks wie in Abschnitt 5.2 um *Kernel*-Benchmarks handelt, also um Komponenten eines grösseren Systems, können wir diese Ergebnisse als Indiz dafür nehmen, dass sich auch bei grösseren Systemen die Anzahl der Regionenanweisungen ebenfalls in Grenzen hält.

5.4 Durchführung der Laufzeitversuche

Mit der Durchführung der Benchmarks wollen wir einen Vergleich unserer Methode mit den existierenden Methoden der Speicherverwaltung ziehen. Zunächst beschreiben wir, wie wir die Benchmarks verwendet haben.

5.4.1 Erstellung der Benchmarks

Die in Abschnitt 5.2 beschriebenen Benchmarks haben wir in folgenden Versionen übersetzt:

Die „Boehm GC"-Version Hierbei wurden die Allokationsanweisungen der Programme in Aufrufe an die Allokationsroutine des Boehm-Demers-Weiser-

Speicherbereinigers [BW88] übersetzt, und der Speicherbereiniger zu den Programmen dazugebunden.

Bei dem Boehm-Demers-Weiser-Speicherbereiniger [BW88] handelt es sich um eine reife Implementierung, die nach dem „Markieren und Abräumen"-Prinzip arbeitet. Die Urheber dieser Implementierung, die diese Implementierung auch aktuell noch weiter pflegen und entwickeln, können wir als Experten auf dem Gebiet der Speicherbereinigung ansehen. Die Implementierung selber ist einschlägig bekannt dafür, dass sie nicht nur zuverlässig und vollständig ist, sondern dass sie auch ein hohes Mass an Effizienz besitzt, und dass bei der Implementierung sehr viel Wissen über das tatsächliche Verhalten der Hardwareplatformen, für die sie verfügbar ist, mit eingeflossen ist.

Die Programme dieser Version haben mit dem Speicherbereiniger eine korrekte Speicherverwaltung, so wie wir dies mit unserem Ansatz erreichen. So gesehen haben wir hier einen realistischen Vergleichsfall zu den mit unserer Methode übersetzten Programmen geschaffen.

Allerdings haben wir haben jedoch am Ende von Abschnitt 3.5.3 mit dem Zitat aus [BCR03] illustriert, dass in Echtzeitsystemen sehr viel Rechenzeit für den Speicherbereiniger bereitgestellt werden muss — nicht etwa, weil er einen so hohen Laufzeitbedarf hat, sondern weil seine Laufzeit so schwer vorherzusagen ist. Die Programme der „Boehm GC"-Version laufen jedoch nicht unter Echtzeit-Bedingungen. Die Laufzeit, die wir angeben, ist schlussendlich nur die Summe der Laufzeit, die das Programm selber benötigt hat, und der Laufzeit, die für den Speicherbereiniger tatsächlich angefallen ist.

Die „No Mmgt"-Version Für diese Version wurden die Allokationsanweisungen der Programme in Aufrufe an die `malloc`-Anweisung [Kam98a] übersetzt.

Die Benchmarks liegen als Java-Programme vor, und enthalten deshalb keine Anweisungen, um auf der Halde allozierte Objekte zu löschen. Bei dieser Version wird also nur Speicher auf der Halde neu alloziert, und nie wieder freigegeben. Damit haben wir bei dieser Version keine Speicherverwaltung, die im Sinne von Abschnitt 4.2 korrekt ist. Um eine Korrektheit in diesem Sinne zu erreichen, müssten noch Anweisungen hinzugefügt werde, die an geeigneten Stellen den Objektgraphen traversieren und Objekte deallozieren. Damit würde die Laufzeit weiter ansteigen. Diese Version stellt damit für die Ausführung der Benchmarks mit expliziter Speicherverwaltung eine untere Grenze dar.

Die „Regions"-Version Für diese Version wurden die Programme mit der in Abschnitt 4 beschriebenen Methode übersetzt. Die Lebenszeiten von Objekten fangen wir so ein, wie in Kapitel 4 beschrieben. Im Unterschied zu der am Anfang von

Abschnitt 4.5.2 beschriebenen Vorgehensweise fangen wir die Lebenszeiten nicht durch Dominator und Postdominator ein, sondern immer mit Anfang und Ende einer Prozedur. Um Konflikte zwischen Regionen aufzulösen, verwenden wir Referenzzähler.

Parameter der Benchmarks Die Parameter, mit denen die Benchmarks aufgerufen wurden, sind in Tabelle 5.3 aufgeführt.

Benchmark	Parameter
BH	-b 2000 -m
BiSort	-s 2500000 -m
Em3d	-n 100000 -d 10 -i 10 -m
Health	-l 9 -t 15 -s 1010 -p -m
MST	-v 2048 -m
Perimeter	-l 16 -p -m
Power	-m
TreeAdd	-l 23 -m
TSP	-c 50000 -m
Voronoi	-n 200000 -m

Tabelle 5.3: Parameter der Benchmarks

5.4.2 Laufzeitbedarf

Mit der Speicherverwaltung durch Regionen werden zum einen zur Laufzeit mehr Instruktionen ausgeführt. Zum anderen ist die Allokation von Objekten in einer Region etwas schneller als die Allokation auf der Halde, sei es durch eine Freispeicherverwaltung mit expliziter Speicherfreigabe, oder durch die Freispeicherverwaltung eines Speicherbereinigers.

In Abschnitt 5.3 haben wir festgestellt, dass sich die statische Anzahl dieser Anweisungen in Grenzen hält. In diesem Abschnitt messen wir nach, welchen Einfluss sie auf die Laufzeit haben.

Angabe der Laufzeiten In Tabelle 5.4 geben wir die Laufzeitergebnisse der beschriebenen Versionen an. Wir vergleichen die Laufzeitergebnisse der „Boehm GC"- und der „No Mmgt"-Version mit der „Regions"-Version, indem wir zusätzlich das Verhältnis der Laufzeiten von der „Boehm GC"- bzw. von der „No Mmgt"-Version zu der „Regions"-Version angeben. Wir bilden das Verhältnis so, dass Werte ≤ 1 bedeuten, daß die „Regions"-Version eine kürzere Laufzeit als die Vergleichs-Version hat.

Methodik der Messung Die Laufzeiten wurden in allen Fällen durch Auslesen der Benutzer-Zeit (*User Time*) des Betriebssystems erhalten. Dadurch enthalten sie ausschließlich die Rechenzeit, die für das abarbeiten der Benchmarks nötig war. Wartezeiten für Ein-/Ausgabeaktivitäten und Zeiten, die auf Geheiß des Prozesses innerhalb des Betriebssystemkerns verbraucht wurden, sind nicht enthalten. Um die Zeiten in Tabelle 5.4 zu erhalten, wurden die einzelnen Benchmarks fünf Mal gestartet; aus den einzelnen Laufzeiten wurde das arithmetische Mittel gebildet.

Benchmark	**Laufzeit** **absolut** (s)			**relativ** zu t_{Regs}	
	t_{NoMmgt}	t_{GC}	t_{Regs}	$t_{\text{Regs}}/t_{\text{No Mmgt}}$	$t_{\text{Regs}}/t_{\text{GC}}$
`BH`	6.46	9.04	9.68	1.50	1.07
`BiSort`	27.67	31.75	27.81	1.01	0.88
`Em3d`	7.93	16.57	7.67	0.97	0.46
`Health`	6.33	15.92	4.17	0.66	0.26
`MST`	7.66	10.25	7.41	0.97	0.72
`Perimeter`	1.03	1.72	0.89	0.86	0.52
`Power`	4.85	5.66	7.07	1.46	1.25
`TreeAdd`	3.56	6.29	1.74	0.49	0.28
`TSP`	0.91	1.08	0.87	0.96	0.81
`Voronoi`	7.07	11.83	6.18	0.87	0.52

Tabelle 5.4: Laufzeiten der Benchmarks

Bewertung Aus den Werten von Tabelle 5.4 entnehmen wir, dass die „Regions"-Version in sieben der zehn Fälle unter der Laufzeit der „No Mmgt"-Version und in acht der zehn Fälle unter der Laufzeit der „Boehm GC"-Version liegt. In zwei Fällen liegt sie deutlich über der Laufzeit der „No Mmgt"-Version und in einem Fall deutlich über der Laufzeit der „Boehm GC"-Version.

Die „Regions"-Version kann also mit ihrer Laufzeit nicht nur mit der „Boehm GC"-Version, sonder sogar mit der „No Mmgt"-Version gut mithalten. Wir profitieren hier von der Tatsache, dass die Allokation in einer Region etwas weniger Laufzeit kostet als die Allokation eines Objektes auf der Halde. Den zusätzlichen Aufwand für das Erstellen und Verwerfen von Regionen können wir damit mehr als kompensieren.

5.4.3 Speicherverbrauch

In diesem Abschnitt betrachten wir das Verhalten unseres Ansatzes in Bezug auf den Speicherverbrauch. Wir vergleichen den Speicherverbrauch der „Regions"-Version

mit dem der „Boehm GC"-Version. Das Verhalten der „No Mmgt"-Version ist trivial und wird im weiteren nicht dargestellt.

Überblick Den absoluten Speicherbedarf und die durchschnittlichen Objektgrössen geben wir in Tabelle 5.5 an. Für die Benchmarks (in der „Regions"-Version) ergaben sich die Allokationsraten in Tabelle 5.6.

	Speicherbedarf		**Durchschnittl.**
Benchmark	Objekte	kByte	**Objektgrösse** (Byte)
BH	10211226	147904.21	14.83
BiSort	2097154	32768.02	16.00
Em3d	1400036	46094.15	33.71
Health	11426594	146455.37	13.12
MST	8394757	102464.04	12.50
Perimeter	452924	12384.50	28.00
Power	1544407	21229.05	14.08
TreeAdd	8388610	131072.02	16.00
TSP	65538	2559.99	40.00
Voronoi	5985234	107513.19	18.39

Tabelle 5.5: Speicherbedarf und durchschnittl. Objektgrössen der Benchmarks

	Allokationsrate	
Benchmark	Objekte/s	kByte/s
BH	288452.71	4178.08
BiSort	907858.87	14185.29
Em3d	313207.16	10311.89
Health	2539243.11	32545.64
MST	3850805.96	47001.85
Perimeter	348403.08	9526.54
Power	131438.89	1806.73
TreeAdd	23301694.44	364088.93
TSP	211412.90	8258.04
Voronoi	5810906.80	104381.74

Tabelle 5.6: Allokationsraten der Benchmarks

Methodik der Messung Um zu quantifizieren, wie die Benchmarks mit dem Haldenspeicher umgehen, wurden Speicheranforderungen und -rückgaben während eines Laufes protokolliert.

Für die „Boehm GC“-Version wurde durch Auslesen der Datenstrukturen der Programmbibliothek protokolliert, zu welchen Zeitpunkten der Speicherbereiniger den Speicher neu organisiert, und wann er dabei die Halde vergrössert. Aus diesen Daten können folgendermaßen Rückschlüsse auf das Verhalten des Programmes gezogen werden: Der Speicherbereiniger vergrössert die Halde immer dann, wenn von der existierenden Halde kein Speicher freigegeben werden kann, und zu wenig Speicher vorhanden ist, um eine anstehende Speicheranfrage zu beantworten (siehe hierzu auch die Diskussion in Abschnitt 3.5.1). Die Halde wird dabei nie verkleinert, auch wenn sich der vom Mutator belegte Speicher wieder reduziert. Solange der Speicherbereiniger also die Halde nach einer Reorganisation vergrössert, können wir darauf schließen, dass er so viele Objekte, die auf der bisherigen Halde alloziert sind, als erreichbar erkannt hat, dass er nicht ausreichend viele Objekte freigeben kann. Wenn nach einer Reorganisation die Halde nicht vergrössert wird, so ist durch die Reorganisation offenbar ausreichend viel Speicher der Halde als frei erkannt worden. Wir erfahren dabei jedoch nicht, wieviel Speicher noch als belegt betrachtet wird, oder wieviel Speicher nun zur Beantwortung nachfolgender Speicheranfragen zur Verfügung steht.

In den Diagrammen führen wir für die „Boehm GC“-Version auf, wieviel Speicher der Speicherbereiniger für die Halde verwendet, und zusätzlich, wieviel Speicher aufgrund der Konservativität des Boehm-GCs nicht freigegeben werden kann, und wieviel Speicher zusätzlich für die Objektmarkierungen bei der Traversierung des Objektgraphen verbraucht wird. Nur bei manchen Benchmarks ergibt sich hier ein deutlicher Unterschied zwischen Halde und Mehrverbrauch, während bei den meisten Benchmarks der Unterschied gering oder kaum zu erkennen ist. Auf dem Graphen für die Grösse der Halde zeigen wir zusätzlich noch, zu welchen Zeitpunkten der Speicherbereiniger den Objektgraphen traversiert und neu organisiert.

Bei der „Regions“-Version liefert uns eine Instrumentierung der Anweisungen Auskunft, die Regionen anlegen und verwerfen, und die Objekte in Regionen anlegen.

Zunächst fassen wir charakteristische Daten sowohl der „Boehm GC“-Version als auch der „Regions“-Version in Tabellen 5.7 und 5.8 zusammen. Vergleichen wir Tabelle 5.4 und Tabelle 5.8, so stellen wir fest, dass wir nur bei `BH` und bei `Power` feststellen können, dass eine hohe Allokationsrate von Regionen korreliert mit einer erhöhten Laufzeit im Vergleich mit der „No Mmgt“-Version oder der „Boehm GC“-Version. Diese Beobachtung suggeriert, dass es von Nutzen sein könnte, die Zugriffsbereiche anhand geeigneter Heuristiken zu vergröbern, um damit die Anzahl der Regionenoperationen zu mindern und gleichzeitig ähnliche Laufzeitvorteile wie bei den verbleibenden Benchmarks zu erreichen.

Benchmark	**Reorganisationen der Halde**	**Vergrösserungen der Halde**
BH	238	20
BiSort	63	37
Em3d	383	44
Health	93	37
MST	97	66
Perimeter	47	25
Power	156	14
TreeAdd	35	19
TSP	96	65
Voronoi	96	41

Tabelle 5.7: Verhalten des Speicherbereinigers

	Regionen		**Allokationsrate**
Benchmark	Anzahl	Speicher (kBytes)	1/s
BH	3033147	130330.54	85682.12
BiSort	11	0.47	4.76
Em3d	42	1.80	9.40
Health	87420	3756.33	19426.67
MST	20	0.86	9.17
Perimeter	20	0.86	15.38
Power	1520020	65313.36	129363.40
TreeAdd	13	0.56	36.11
TSP	11	0.47	35.48
Voronoi	21	0.90	20.39

Tabelle 5.8: Verhalten der Regionen

Benchmark	**Regionen**	**Boehm GC**
	$M_{\text{final}}/M_{\text{alloc}}$	
BH	0.8299	0.0258
BiSort	1.0000	1.2070
Em3d	0.7627	1.3315
Health	0.2006	0.2817
MST	1.0000	1.1970
Perimeter	1.0000	1.3270
Power	0.0211	0.0427
TreeAdd	1.0000	1.1440
TSP	1.0000	1.4917
Voronoi	1.0000	0.4302

Tabelle 5.9: Speicherbelegung im Vergleich zum angeforderten Speicher

Das Verhalten der Benchmarks im Einzelnen Die Werte, die wir durch die oben genannte Instrumentierung gemessen haben, tragen wir ab über der laufenden Summe des bisher angeforderten Speichers. Damit blenden wir zum einen Details Laufzeitverhalten der Benchmarks (wie lang anhaltende Berechnungen in einzelnen Phasen der Programmausführung, die keine Haldenoperationen beinhalten) aus. Zum anderen können wir damit die Messwerte der unterschiedlichen Versionen überlagern, indem wir nur die Ordinaten-Skala in Übereinstimmung bringen. Damit können wir das Verhalten der Versionen vergleichen.

In den Abbildungen 5.1 bis 5.10 tragen wir über dem angeforderten Speicher jeweils den belegten Speicher der „Boehm GC"-Version und der „Regions"-Version ab. Für die „Boehm GC"-Version tragen wir zusätzlich ab, wieviel Speicher der Speicherbereiniger durch scheinbare Zeiger verliert, und wieviel Speicher er zusätzlich für die Markierung der Objekte benötigt. In vielen Fällen, zum Beispiel in Abb. 5.2, ist dieser Mehrbedarf allerdings so gering, dass er in den Abbildungen nicht zu erkennen ist. Auf dem Niveau der Halde des Speicherbereinigers markieren wir zusätzlich die Zeitpunkte, zu denen der Speicherbereiniger die Halde neu organisiert hat.

BH Das Verhalten von BH in Abbildung 5.1 zeigt, dass die „Regions"-Version wesentlich mehr Speicher verbraucht, als die „Boehm GC"-Version. Der Speicherbereiniger erkennt häufig, dass nur wenige allozierte Objekte erreichbar sind, und kann häufig weitere Speicheranforderungen mit der Halde beantworten. Betrachtet man zusätzlich noch die Laufzeiten dieses Benchmarks in Tabelle 5.4, so erkennt man, dass für dieses Programm der Speicherbereiniger unserer Methode klar überlegen ist, selbst mit der Abschätzung, die sich in [BCR03] ergeben würde.

BiSort In Abbildung 5.2 sieht man, dass die beiden Versionen von `BiSort` ein vergleichbares Speicherverhalten haben. Während die Grösse der Halde der „Boehm GC"-Version um 20.70% grösser ist als die insgesamt angeforderte Speichermenge, verbraucht die „Regions"-Version weniger Speicher und kann innerhalb einer kleineren Halde ablaufen. Was den Laufzeitbedarf in Tabelle 5.4 angeht, sind sich beide Versionen ebenbürtig.

Em3d In Abbildung 5.3 zeigt `Em3d` ein ähnliches Verhalten wie `BiSort`. Die „Regions"-Version schliesst den Lauf mit einer Halde ab, die 76.27% des allozierten Speichers ausmacht, während die „Boehm GC"-Version mit 133.15% der allozierten Speichermenge deutlich mehr Speicher braucht, als das Programm eigentlich alloziert.

Health Abbildung 5.4 zeigt, dass die „Regions"-Version von `Health` deutlich weniger Speicher braucht (20.06% in Tabelle 5.9) als die „Boehm GC"-Version (28.17%). Im Vergleich zu allen anderen Schaubildern sieht man hier sehr deutlich, an welchen Stellen die Regionen verworfen wurden, während diese Details in den anderen Schaubildern in der graphischen Auflösung untergehen.

Mit einem Laufzeitverhältnis von 0.26 ist die „Regions"-Version ungefähr viermal so schnell wie die „Boehm GC"-Version.

MST Bei dem Lauf der „Boehm GC"-Version von `MST` stellte sich heraus, dass die „Boehm GC"-Version von `MST` zwar noch mit den Parametern aus Tabelle 5.3 innerhalb des zur Verfügung stehenden Hauptspeichers ablaufen konnte, dass jedoch die Instrumentierung, mit der die Daten für Abbildung 5.5 ausgelesen wurden, den Speicherbedarf dermaßen erhöhten, dass dieser Lauf vorzeitig abgebrochen werden musste. Man erkennt, dass die „Regions"-Version innerhalb des zur Verfügung stehenden Hauptspeichers eine grössere Halde erstellen kann als die „Boehm GC"-Version, und ihren Lauf normal beendet.

Perimeter Den Lauf von `Perimeter` absolviert die „Regions"-Version, wie in Abbildung 5.6 gezeigt wird, mit einer deutlich kleineren Halde und, wie bereits in Tabelle 5.4 gezeigt, in wenig mehr als der Hälfte der Zeit.

Power Bei dem Lauf von `Power` bleibt nach einer kurzen Aufbauphase die Anzahl an allozierten Objekten und damit der belegte Speicher innerhalb kleiner Variationen konstant. Die „Boehm GC"-Version findet hier jedoch ungefähr doppelt so viel Speicher als erreichbar als die „Regions"-Version. Durch die hohe Rate, mit der

die „Regions"-Version Regionen anlegt und verwirft, braucht sie die in Tabelle 5.4 gezeigte höhere Laufzeit.

TreeAdd Bei `TreeAdd` ergab sich wie bereits bei `MST`, dass die zur Messung instrumentierte „Boehm GC"-Version nicht innerhalb des zur Verfügung stehenden Hauptspeichers ablaufen konnte, und abgebrochen werden musste. Tabelle 5.9 zeigt, dass der Hauptspeicherbedarf um 49.17% über die Menge an alloziertem Speicher herausgeht. Der Verwaltungsaufwand für diesen Überhang erklärt auch die ungefähr drei Mal so hohe Laufzeit der „Boehm GC"-Version.

TSP Aus Tabelle 5.4 und Abbildung 5.9 entnehmen wir, dass `TSP` in der „Regions"-Version sowohl weniger Hauptspeicher als auch weniger Laufzeit benötigt als in der „Boehm GC"-Version.

Voronoi In Abbildung 5.10 sehen wir, analog zu `Health` in Abbildung 5.4, dass der Speicherbereiniger bei der „Boehm GC"-Version nach einer kurzen Anfangsphase ständig Objekte als nicht mehr erreichbar identifizieren kann, so dass in diesem Fall die Halde deutlich kleiner bleiben kann als bei der „Regions"-Version. Letztere absolviert den Lauf in etwas mehr als der Hälfte der Zeit.

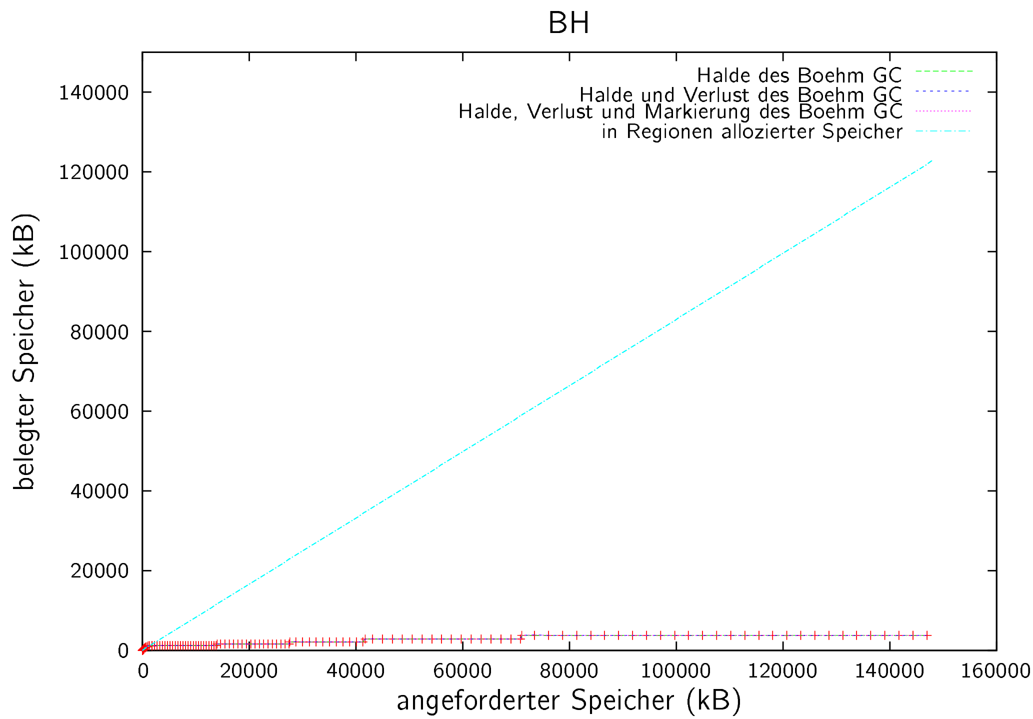


Abbildung 5.1: Speicherverbrauch von `BH`

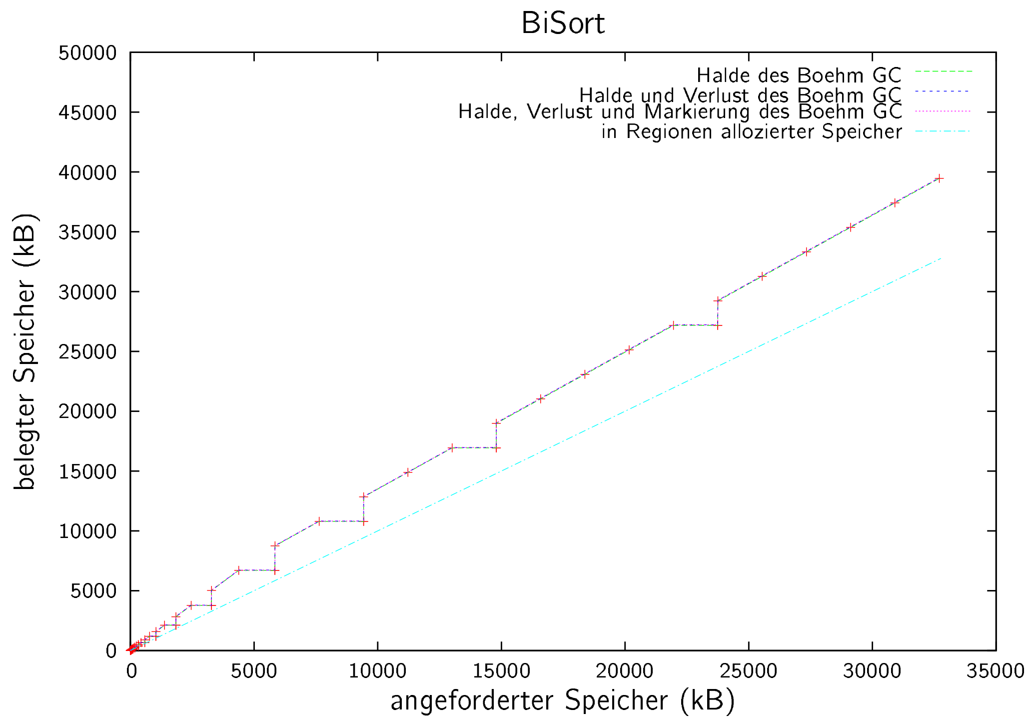


Abbildung 5.2: Speicherverbrauch von `BiSort`

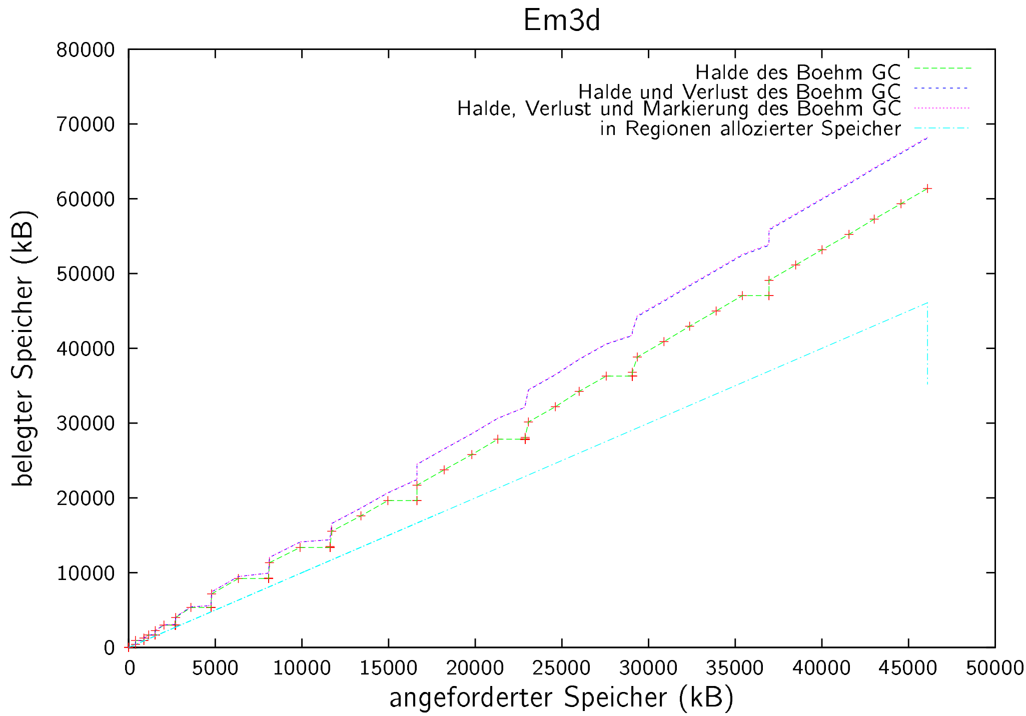


Abbildung 5.3: Speicherverbrauch von `Em3d`

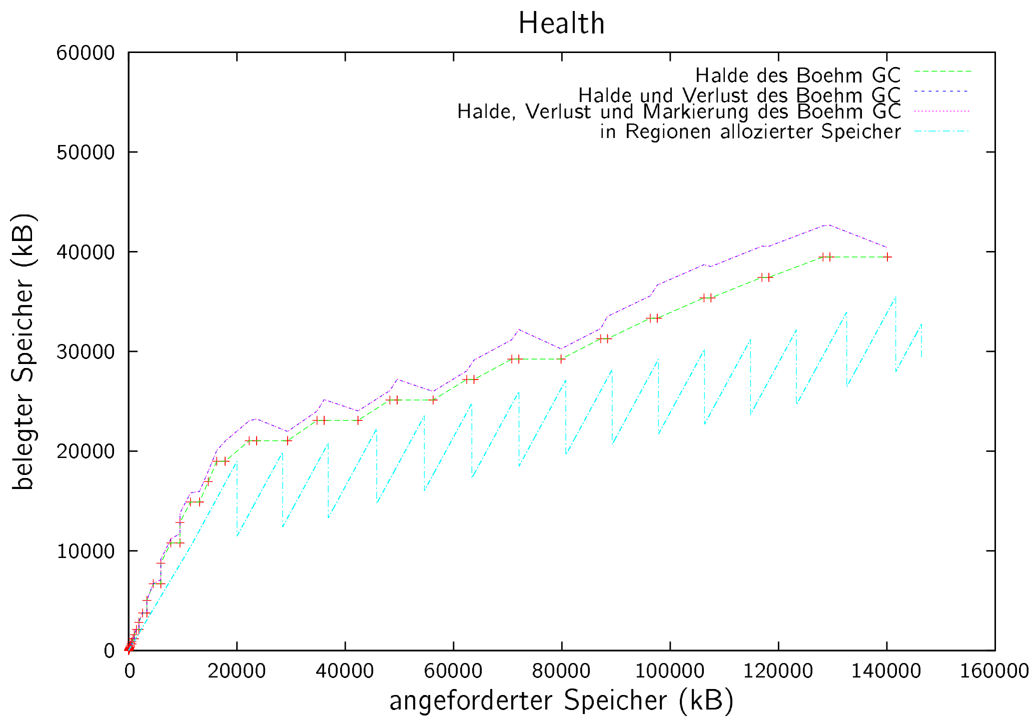


Abbildung 5.4: Speicherverbrauch von `Health`

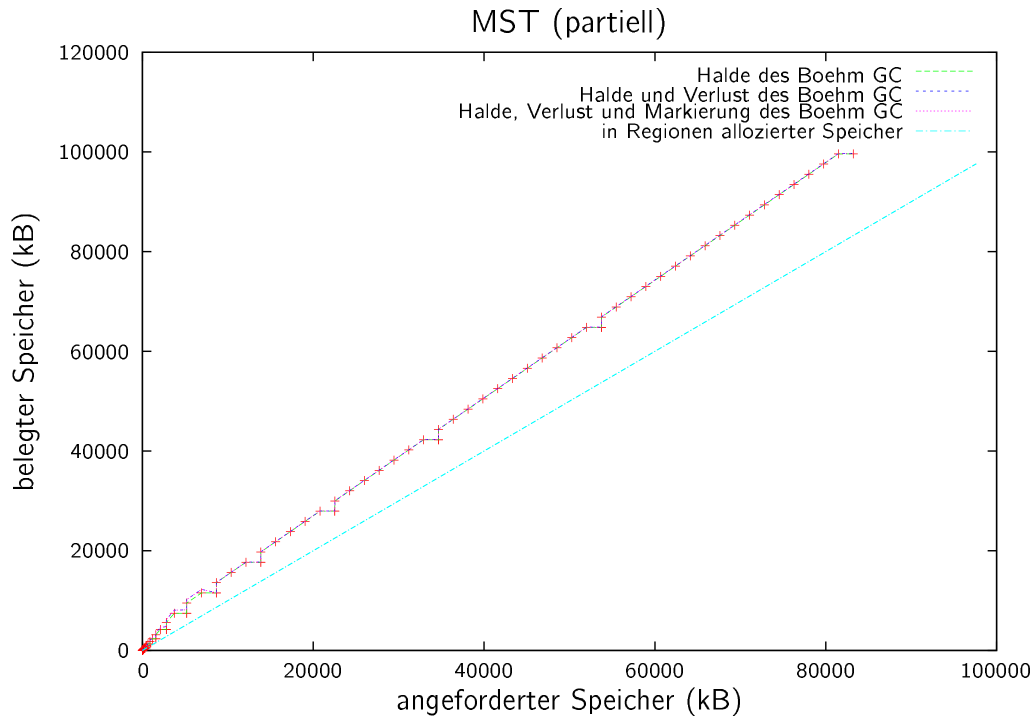


Abbildung 5.5: Speicherverbrauch von `MST`

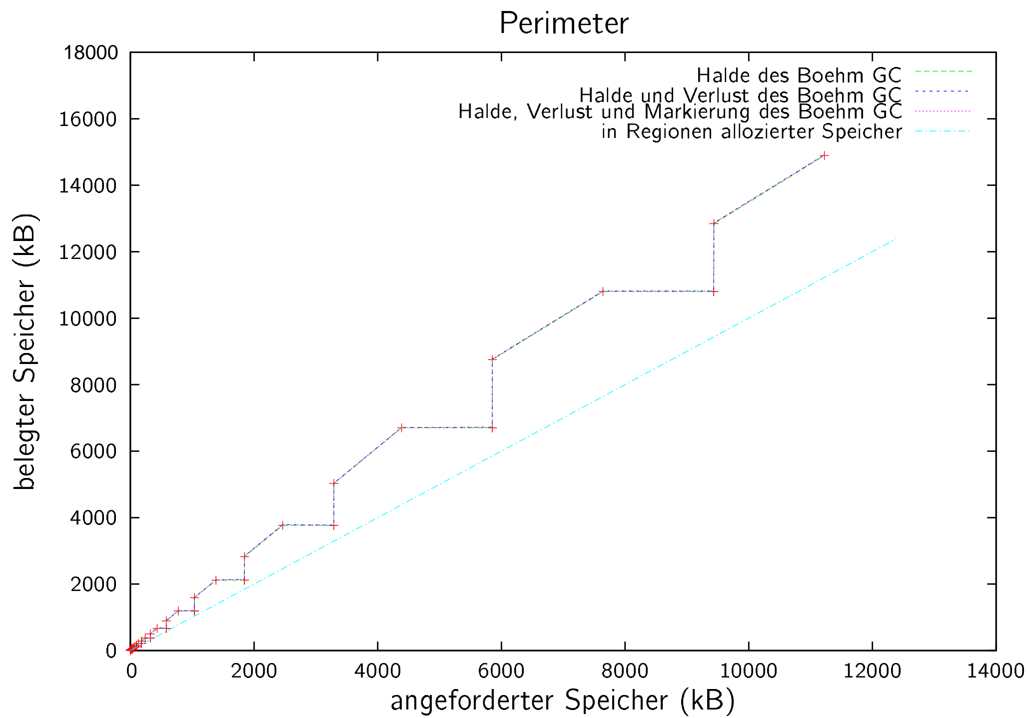


Abbildung 5.6: Speicherverbrauch von `Perimeter`

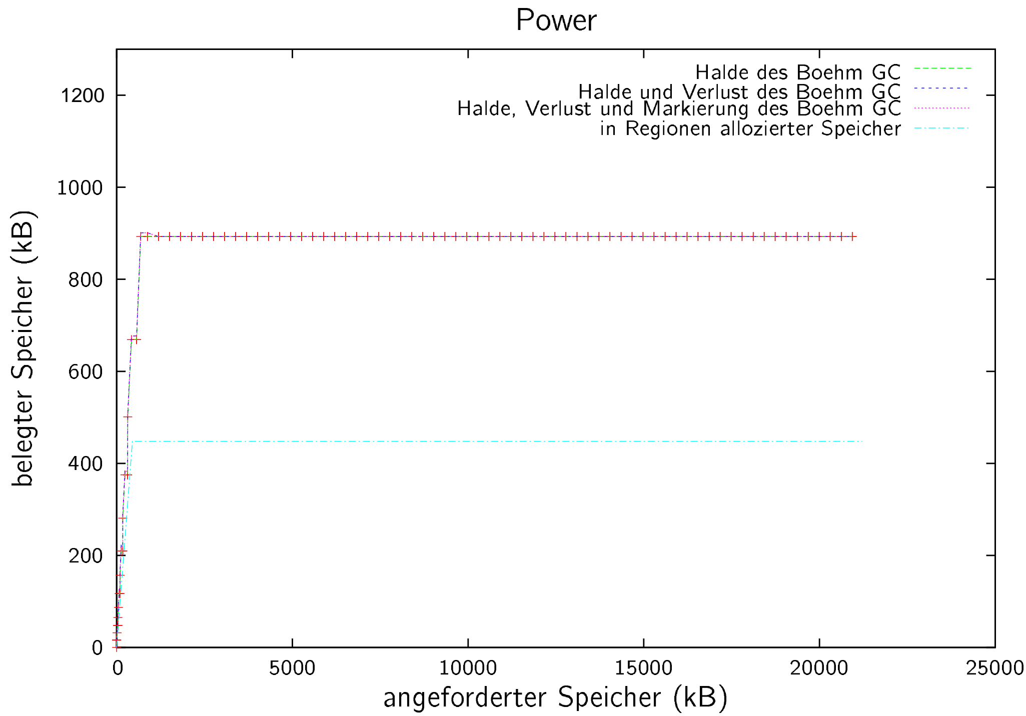


Abbildung 5.7: Speicherverbrauch von `Power`

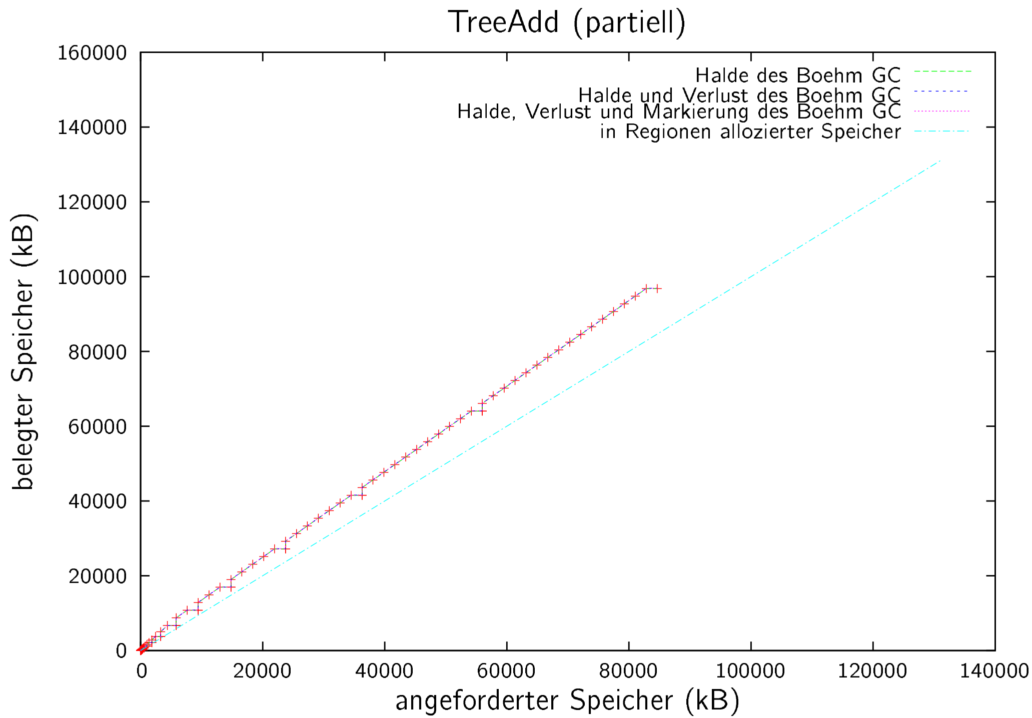


Abbildung 5.8: Speicherverbrauch von `TreeAdd`

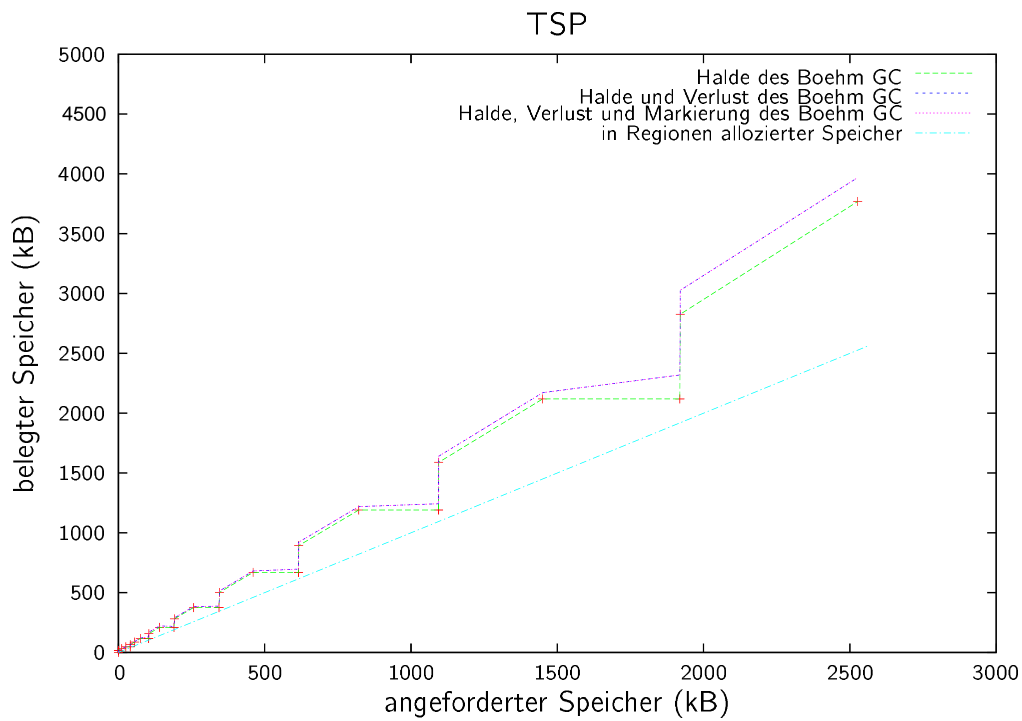


Abbildung 5.9: Speicherverbrauch von `TSP`

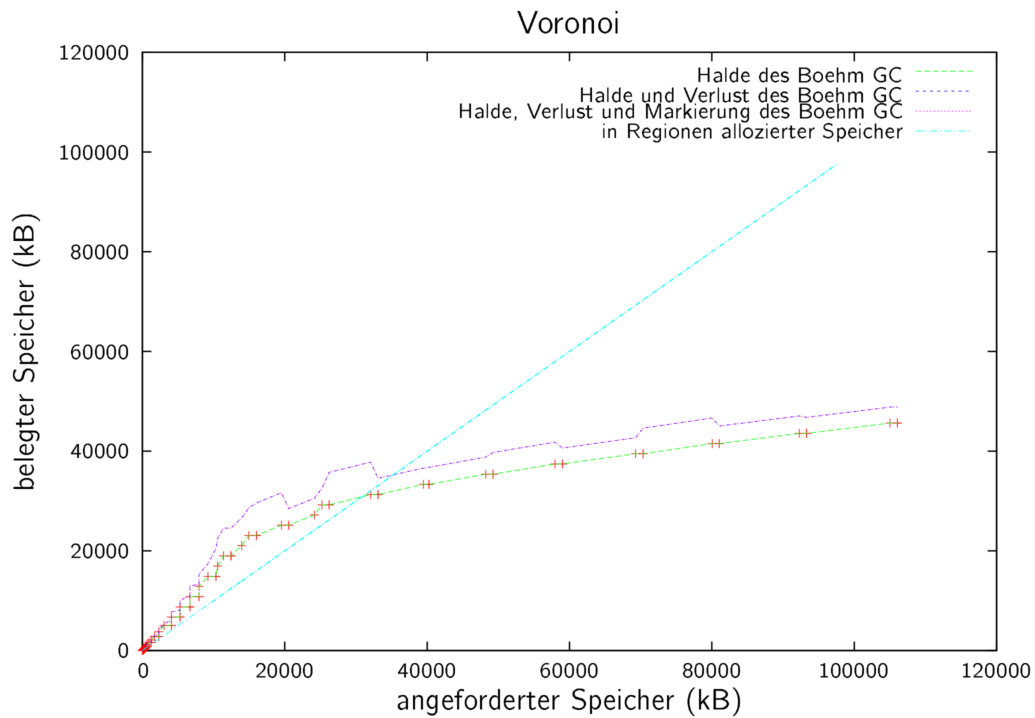


Abbildung 5.10: Speicherverbrauch von `Voronoi`

Kapitel 6

Rückschau und Ausblick

„Die Wissenschaft fängt eigentlich erst da an interessant zu werden, wo sie aufhört.“
Justus von Liebig, „Chemische Briefe“

6.1 Rückschau

Wir haben in dieser Arbeit mit unserem Ansatz eine neuartige Methode der Speicherverwaltung in imperativen Programmen beschrieben, die in prozeduralen oder objekt-orientierten Sprachen implementiert sind.

Unseren Ansatz haben wir erarbeitet in dem Kontext von Systemen, die Echtzeitanforderungen erfüllen müssen, und die in einer sicherheitskritischen Rolle eingesetzt werden sollen.

Wir haben existierende Ansätze betrachtet, und sowohl ihre Vorteile als auch ihre Nachteile aufgezeigt. Dabei haben wir insbesondere die Eignung der existierenden Ansätze für den Einsatz bei echtzeitfähigen Systemen in sicherheitskritischen Rollen betrachtet. Wir haben anhand der existierenden Ansätze herausgearbeitet, welche Anforderungen eine Speicherverwaltung erfüllen muss, damit sie sich von den existierenden Ansätzen positiv abhebt.

Wir haben uns einen expliziten Begriff davon geschaffen, was wir unter der korrekten Nutzung des Haldenspeichers verstehen.

Wir haben unseren Ansatz so konstruiert, dass er voll automatisch durch den Übersetzer ausgeführt wird. Der Ansatz erfordert weder eine Einschränkung der Ausdrucksmächtigkeit der Quellsprache, noch erfordert er eine Erweiterung der Sprache. Er bürdet dem Programmierer keinerlei zusätzliche Disziplin bei der Erstellung der Software auf.

Wir haben unser Ziel dadurch erreicht, dass wir das Programm durch den Übesetzer analysieren lassen, und dass wir basierend auf den Ergebnissen der Ana-

lyse Anweisungen in das Programm einfügen, die die nötigen Aufgaben der Speicherverwaltung übernehmen.

Dadurch, dass wir uns auf die Ergebnisse einer Programmanalyse verlassen, eliminieren wir die Möglichkeit, dass sich durch die Arbeit des Programmierers ein Programm ergibt, das den Haldenspeicher nicht korrekt nutzt.

Basierend auf unserem Begriff über die korrekte Nutzung des Haldenspeichers, und mit den Informationen der Programmanalyse — der Zeigeranalyse und der Zugriffsanalyse — haben wir den Begriff der Zugriffsbereiche geschaffen, mit denen wir die Lebenszeiten von auf der Halde allozierten Objekten einfangen.

Wir haben eine Reihe von Benchmarks mit unserem Ansatz erstellt, und wir haben sie mit anderen Versionen verglichen. Wir haben gezeigt, dass die Ergebnisse, die unser Ansatz liefert, durchaus konkurrenzfägig sind, sowohl in Begriffen der Laufzeit, als auch in Hinblick auf den Speicherverbrauch.

Mit unserem Ansatz setzten wir an den durch die Analyse bestimmten Stellen Anweisungen ein, die Regionen anlegen und verwerfen. Wenn eine Region verworfen wird, werden dabei alle Objekte dealloziert, die innerhalb dieser Region alloziert waren. Der Laufzeitaufwand, der durch diese Anweisungen verursacht wird, entsteht an festen Stellen des Programmes, nicht durch einen Laufzeitmechanismus. Damit ist der Laufzeitbedarf des Programmes, in das die Regionen-Anweisungen eingesetzt wurden, für eine Laufzeitanalyse, wie sie bei der Überprüfung zur Sicherstellung der Echtzeitfähigkeit eines Programmes nötig ist, transparent. Genauso transparent stellt sich der Speicherverbrauch des resultierenden Programms bei einer Echtzeit-Analyse dar.

6.2 Kritik

Vollautomatischer Ansatz Wir haben im letzen Abschnitt betont, dass wir mit unserem Ansatz immer Programme schaffen, deren Speicherverwaltung korrekt (in dem von uns definierten Sinne) sind, ohne dass ein Aufwand seitens des Programmierers erforderlich ist. Wir schliessen aus, dass der Programmierer Fehler machen kann.

Mit einem solchen vollautomatischen Ansatz nehmen wir allerdings auch dem Programmierer die Möglichkeit, „etwas richig zu machen“. Insbesondere ist es für den Progammierer nicht möglich, in einem Programm eine Speicherverwaltung zu implementieren, die besser als das ist, was unser Ansatz zu leisten vermag. Während unser Ansatz in Punkto Laufzeit wenig zu wünschen lässt, haben wir in Abschnitt 5.4.3 zur Kenntniss nehmen müssen, dass wir mit unserem Ansatz manchmal deutlich mehr Speicher brauchen, als dies die Programme mit einem automatischen Speicherbereiniger tun.

Wenn wir die Regionen-Anweisungen in das Programm einsetzen, setzen wir sie an Stellen ein, die von unserer Programmanalyse bestimmt wurden. Damit fällt der Laufzeitaufwand für die Ausführung dieser Anweisungen — im Gegensatz zu den Speicherbereinigern — an einer festen Stelle an. Aber wir können dem Programmierer kein Mittel in die Hand geben, die Platzierung dieser Anweisungen in irgendeiner Art zu beeinflussen. Der Programmierer kann sein Wissen über das Verhalten des Programmes, das er ja schliesslich selber geschrieben hat, nicht mit einbringen.

Abhängigkeit von der Zeigeranalyse Unsere Ergebnisse, insbesondere die Zugriffsbereiche, können nur so gut sein, wie die Analysedaten, die uns zur Verfügung stehen. Insbesondere hängt das Ergebnis kritisch davon ab, wie gut die Zeigeranalyse die Haldenoperationen analysieren kann. Die Erstellung einer Zeigeranalyse sehen wir als ein Problem, dessen prinzipielle Lösung uns zwar fertig vorliegt [Hin01], dessen technische, ingenieursmässige Umsetzung jedoch eine anspruchsvolle Herausforderung ist [AW93, SRW96, Tra00, HDH04]. Wir können in unserer Arbeit keine Angaben darüber machen, in wie weit unsere Ergebnisse durch die Analyse bestimmt sind, die wir für die praktische Umsetzung unseres Ansatzes geschaffen ahben, und wir können nicht abschätzen, welche Ergebnisse wir mit einer anderen Zeigeranalyse erhalten hätten.

Keine Deallokation von einzelnen Objekten in Regionen Wir haben in Kapitel 5 festgestellt, dass wir mit unserem Ansatz in einigen Fällen einen Speicherverbrauch verursachen, der deutlich über dem liegt, was wir mit einem Speicherbereiniger benötigen. Wir haben dies darauf zurückgeführt, dass in diesen Fällen eine Datenstruktur nicht nur aufgebaut, sonder auch geändert wird, wobei wir mit unserem Ansatz nicht feststellen können, an welchen Stellen einzelne Objekte nicht mehr benötigt werden. Diese Objekte bleiben in den Regionen alloziert, obwohl dies eigentlich nicht nötig wäre, während der Speicherbereiniger diese Objekte regelmässig als frei erkennt und wieder der Freispeicherverwaltung übergibt.

Dass wir innerhalb der Regionen keine Objekte deallozieren können, ist dadurch bedingt, dass wir das Programm nicht genau genug analysieren können. Die Zeigeranalysen, die hier präzise genug wären, sind noch Thema der Forschung [SRW96] und stehen uns leider nicht in verwertbarer Form zur Verfügung.

6.3 Offene Fragen

6.3.1 Kombination mit Speicherbereinigern

Im letzten Absatz des vorangegangenen Abschnittes hatten wir unsere Abhängigkeit von der Qualität der zur Verfügung stehenden Zeigeranalyse angesprochen. Mit der Leistungsfähigkeit der Analyse steigt unser Verständnis von der konkreten Struktur des Objektgraphen, und damit unsere Fähigkeit, Objekte feiner zu differenzieren und schliesslich Objekte möglicherweise früher zu deallozieren. Orthogonal dazu könnte es sich anbieten, unseren Ansatz zu erweitern, indem man zusätzlich einen automatischen Speicherbereiniger einsetzt.

Synergie-Effekte In einem System, das unseren Ansatz mit einem Speicherbereiniger kombiniert, können sich die Vorteile beider Ansätze ergänzen: Innerhalb von Gruppen, die die Analyse nicht unterscheiden kann, kann der Speicherbereiniger noch erreichbare Objekten von nicht mehr erreichbaren Objekten unterscheiden; semantischer Müll, den der Speicherbereiniger nicht erkennen kann, kann durch unseren Ansatz explizit beseitigt werden. Objekte, die wir mit unserem Ansatz löschen, braucht der Speicherbereiniger nicht zu traversieren, so dass sich seine Laufzeit reduziert. Wir können erwarten, dass mit dieser Kombination zumindest der Speicherbedarf — punktweise — jeweils am Minimum dessen liegt, was jede Methode alleine erreichen kann. Was die Laufzeit angeht, können wir erwarten, dass wir — wenn schon nicht immer — so doch häufig besser abschneiden, als dies ein Speicherbereiniger alleine tun würde.

Massnahmen zur Kooperation Da wir mit unserem Ansatz semantischen Müll erkennen, kann es sein, dass Objekte gelöscht werden, obwohl noch Referenzen auf diese Objekte im Zustand des Programms vorhanden sind. Ein Speicherbereiniger würde diesen Referenzen folgen. Zum einen könnte er dabei auf einen Speicherblock stossen, der bereits dealloziert ist — das könnte man in den Datenstrukturen der Freispeicherverwaltung erkennen. Er könnte zum anderen auf einen Speicherblock stossen, der bereits wieder für ein anderes Objekt alloziert wurde — ein Fall, der aus den Datenstrukturen der Freispeicherverwaltung alleine nicht zu erkennen ist. Ein Objekt, das wir explizit deallozieren, kann also nicht unmittelbar als „frei“ an die Freispeicherverwaltung gemeldet werden, sondern in einem neuen, dritten Zustand „dealloziert, aber womöglich noch referenziert“ vorgehalten werden. Der Speicherbereiniger kann jede Referenz, die auf ein solches Objekt (besser, die in einen solchen Block hinein) zeigt, überschreiben — nach einer vollständigen Traversierung des Speicherbereinigers über den Objektgraphen sind alle Objekte und alle Referenzen

besucht worden, und damit auch alle Referenzen in diese deallozierten Blöcke hinein beobachtet worden.

Es bietet sich intuitiv an, diese Referenzen in deallozierte Blöcke mit dem *Null*-Zeiger (`NULL` oder `0` in `C++`, `nil` in *Lisp* etc.) zu überschreiben. Schließlich wird dies auch vom Programmierer explizit durchgeführt, um eine Referenz als „leer“ zu markieren. Allerdings könnte das Programm mit dem Wert einer solchen Referenz — ohne danach auf das referenzierte Objekt zuzugreifen — Operationen durchführen, die auf den weiteren Verlauf des Programms Einfluss haben; zum Beispiel könnte man eine bedingte Anweisung darauf basieren, wie sich die Referenz zu einer anderen Referenz — oder eben dem *Null*-Zeiger vergleicht.

Ironischerweise ist die einzige Information, die man aus einem solchen Vergleich gewinnen kann, die, ob die Referenz noch ein Haldenobjekt referenziert (so dass man also darauf zugreifen könnte), oder ob sie das gleiche Haldenobjekt referenziert wie eine andere Referenz. Mit dieser Information lässt sich entscheiden, ob ein Zugriff auf ein Objekt möglich ist, oder ob man einen ungültigen Zugriff über einen *Null*-Zeiger durchführen würde.

Nach einer Berechnung, in die eine Referenz eingeht, wird man also in der Regel auch zumindest erwarten, einen Ausführungspfad zu finden, der einen Zugriff auf das möglicherweise referenzierte Objekt durchführt. Durch diesen Zugriff wiederum wäre ausgeschlossen, dass das Objekt vorher dealloziert wird. Alleine die Möglichkeit jedoch, dass ein Programm diesen Objektzugriff dann auslässt, so dass wir mit unserem Ansatz das referenzierte Objekt bereits früher deallozieren, zwingt uns dazu, eine Alternative zum Überschreiben mit dem *Null*-Zeiger zu suchen. Stattdessen könnte man einen anderen Wert benutzen, der ebenfalls von allen Referenzen auf Haldenobjekte unterscheidbar ist, der aber innerhalb der Sprache nicht existiert.

6.3.2 Deterministische Ausführung von Destruktoren

Die Erstellung von Objekten verkapseln Programmiersprachen häufig in sog. Konstruktoren (*Constructors* in *Ada*, `C++` und *Java*, `make`-Features in *Eiffel* und *Sather*). Analog dazu bieten Sprachen, die eine explizite Speicherverwaltung vorsehen, häufig sog. Destruktoren, die bei der Deallokation von Objekten ausgeführt werden. In Sprachen wie *Java* oder *Eiffel*, die ausschlieślich Speicherbereinigung verwenden, ist der Zeitpunkt, zu dem ein Objekt dealloziert wird, häufig nicht deterministisch vorherzusagen, da der Speicherbereiniger erst dann den Speicher neu zu organisieren braucht, wenn die Freispeicherverwaltung über zu wenig Speicher verfügt, um Speicheranforderungen zu beantworten.

In *Java* existiert das Konzept des sog. *Finalizers* [GJS97, §12.6], eine Methode, die dann an einem Objekt aufgerufen wird, wenn der Speicherbereiniger dieses Objekt wieder der Freispeicherverwaltung übergibt.

Begründet wird dieses Konzept damit, dass Ressourcen, die durch ein Objekt repräsentiert werden, bei der Ausführung des *Finalizers* freigegeben werden können. Beispielsweise könnte eine offene Datei geschlossen werden. Da der Speicherbereiniger jedoch dann den Speicher neu organisiert, wenn die Ressource „Speicher" knapp wird, nicht etwa wenn andere Ressourcen knapp werden, ist durch dieses Konzept nicht garantiert, dass die so verwalteten Ressourcen immer früh genug freigegeben werden, um einen Mangel zu verhindern.

Da wir mit unserem Ansatz Objekte explizit deallozieren, schaffen wir damit auch einen statisch definierten Punkt in der Programmausführung, zu dem der Speicher der Objekte wieder freigegeben wird. Durch die Konstruktion unseres Ansatzes erreichen wir zudem, dass dies der *frühestmögliche* Zeitpunkt ist, zu dem die Objekte dealloziert werden dürfen. Dieser Punkt wird allerdings durch die Ergebnisse unserer Analyse bestimmt; sollte dieser Punkt in einem speziellen Fall an einer ungünstigen Stelle zu liegen kommen, ergibt sich keine Möglichkeit, korrigierend einzugreifen.

Mit der Deallokation eines Objektes kann man nun den *Finalizer* des Objektes aufrufen. Damit erreicht man zum einen, dass die damit verwalteten Ressourcen zum einen ebenfalls frühestmöglich freigegeben werden, zum anderen werden die Ressourcen zu einem statisch festgelegten Punkt in der Programmausführung freigegeben. Die Gefahr, dass diese Ressourcen deshalb ausgehen, weil der Speicherbereiniger den Speicher nicht früh genug reorganisiert, wird damit zumindest vermindert.

Literaturverzeichnis

[Ada87] *Ada 83 Reference Manual and Rationale*, 1987. ANSI/MIL-STD-1815A-1983, ISO 8652:1987.

[Ada95] *Ada 95 Reference Manual and Rationale*, 1995. ISO/IEC 8652:1995.

[AFL95] Aiken, Alexander, Manuel Fähndrich und Raph Levien: *Better static memory management: Improving region-based Analysis of higher-order Languages*. In: *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, Seiten 174–185, New York, NY, USA, 1995. ACM Press.

[AH00] Aycock, John und Nigel Horspool: *Simple Generation of Static Single-Assignment Form*. In: Watt, D.A. (Herausgeber): *CC 2000/ETAPS 2000*, Band 1781/2000, Seite 110 ff, Berlin, Germany, March/April 2000. Springer-Verlag, Berlin. ISSN: 0302-9743.

[App98] Appel, Andrew W.: *SSA is Functional Programming*. SIGPLAN, 33(4):17–20, April 1998.

[AW93] Aßmann, Uwe und Markus Weinhardt: *Interprocedural Heap Analysis for Parallelizing Imperative Programs*. In: Giloi, W. K., S. Jähnichen und B. D. Shriver (Herausgeber): *Programming Models For Massively Parallel Computers*, Seiten 74–82. IEEE Press, Washington, DC, September 1993.

[Bak78] Baker, Henry G.: *List Processing in Real Time on a Serial Computer*. Comm. of the ACM, 21(4):1–13, April 1978.

[Bak92] Baker, Henry G.: *The Treadmill: Real-Time Garbage Collection without Motion Sickness*. SIGPLAN, 27(3):66–70, 1992.

[Bar88] Bartlett, Joel F.: *Compacting Garbage Collection with Ambiguous Roots*. Technischer Bericht 88/2, Digital Equipment Corporation, February 1988.

[Bar90] Bartlett, Joel F.: *A generational, compacting collector for C++*. In: Jul, Eric und Niels-Christian Juul (Herausgeber): *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[BBD+00] Bollella, G., B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull und R. Belliardi: *The Real-Time Specification for Java*. Addison-Wesley, Juni 2000.

[BC92] Boehm, Hans-J. und David Chase: *A Proposal for Garbage-Collector C Compilation*. In: *The Journal of C Language Translation*, Band 4, Seiten 126–141. IECC, December 1992.

[BCHS97] Briggs, Preston, Keith D. Cooper, Timothy J. Harvey und L. Taylor Simpson: *Practical Improvements to the Construction and Destruction of Static Single Assignment Form*. Technischer Bericht, Rice University, 1997.

[BCR03] Bacon, David F., Perry Cheng und V. T. Rajan: *A Real-time Garbage Collector with Low Overhead and Consistent Utilization*. Conference Record of the Thirtieth ACM Symposium on Principles of Programming Languages, 38(1):285–298, January 2003.

[Bec99] Beck, Kent: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. ISBN 0201616416.

[BG01] Bohra, Aniruddha und Eran Gabber: *Are Mallocs Free of Fragmentation?* In: *Usenix Annual Technical Conference*, Seiten 105 – 118. USENIX, 2001.

[BGK99] B. Gomes, D. Stoutamire, B. Weisssman und H. Klawitter: *Sather 1.1: Language Essentials*. International Computer Science Institute Berkeley, 1999.

[BH99] Bogda, J. und U. Holzle: *Removing Unnecessary Synchronization in Java*. In: *Proc. of the 14th ACM-SIGPLAN Conf. on Object-Oriented Programming, Languages and Applications (OOPSLA)*, Seiten 35–46, Denver, CO, November 1999.

[BHS95] Briggs, Preston, Tim Harvey und Taylor Simpson: *Static Single Assignment Construction*. Technischer Bericht, Rice University, July 19 1995.

[Bla99] Blanchet, Bruno: *Escape Analysis for Object Oriented Languages. Application to Java^{TM}*. In: *OOPSLA 99*. Assoc. for Comp. Mach., ACM Press, 1999.

[BM94] Brandis, Marc M. und Hanspeter Mössenböck: *Single-Pass Generation of Static Single Assignment form for Structured Languages*. In: *LOPLAS*. ETH Zürich CH, 1994.

[Boe00] Boehm, Hans-J.: *Fast Multiprocessor Memory Allocation and Garbage Collection*. Technischer Bericht PL-2000-165, Hewlett-Packard Laboratories, Internet and Mobile Systems Laboratory 1501, Page Mill Rd., Palo Alto, CA 94304, December 2000.

[Boe02] Boehm, Hans-J.: *Bounding Space Usage of Conservative Garbage Collectors*. In: *POPL*, Portland, OR USA, Januar 2002. ACM ISBN 1-58113-450-9/02/01.

[Boe05] Boesler, Boris: *Codeerzeugung mit Graphersetzung und Lösungsgraphen*. Doktorarbeit, Universität Karlsruhe (TH), 2005.

[BR01] Beebee, William S. und Martin C. Rinard: *An Implementation of Scoped Memory for Real-Time Java*. In: *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, Seiten 289–305, London, UK, 2001. Springer-Verlag, Berlin.

[BS81] Burris, Stanley und H. P. Sankappanavar: *A Course in Universal Algebra*. Springer-Verlag, 1981.

[BS96] Bacon, David F. und Peter F. Sweeney: *Fast static analysis of C++ virtual function calls*. In: *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 324–341. ACM Press, 1996.

[BW88] Boehm, H. und M. Weiser: *Garbage Collection in an Uncooperative Environment*. Software Practice & Experience, 9(18):807–820, September 1988.

[BZM02] Berger, Emery D., Benjamin G. Zorn und Kathryn S. McKinley: *Reconsidering Custom Memory Allocation*. In: *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, Systems, Languages, and Applications*, Seiten 1–12, New York, NY, USA, 2002. ACM Press.

[CB01] Cheng, Perry und Guy E. Blelloch: *A parallel, real-time garbage collector*. In: *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, Seiten 125–136, New York, NY, USA, 2001. ACM Press.

[CC95] Click, Cliff und Keith D. Cooper: *Combining analyses, combining optimizations*. ACM Trans. Program. Lang. Syst., 17(2):181–196, 1995.

[CCQR04] Chin, Wei-Ngan, Florin Craciun, Shengchao Qin und Martin Rinard: *Region inference for an Object-Oriented Language*. SIGPLAN, 39(6):243–254, 2004.

[CFR+91] Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman und F. Kenneth Zadeck: *Efficiently computing static single assignment form and the control dependence graph*. ACM Trans. Program. Lang. Syst., 13(4):451–490, 1991.

[CGS+99] Choi, Jong-Deok, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar und Sam Midkiff: *Escape Analysis for Java*. In: *Proc. of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA99)*, 1999.

[Cli93] Click, Cliff: *From Quads to Graphs: An Intermediate Representation's Journey*. Technischer Bericht, CRPC - The Center for Research on Parallel Computation at Rice University, October 1993.

[Col60] Collins, George E.: *A method for overlapping and erasure of lists.* Comm. of the ACM, 3(12):655–657, 1960.

[Com64] Comfort, W. T.: *Multiword list items*. Commun. ACM, 7(6):357–362, 1964.

[CP93] Click, Cliff und Michael Paleczny: *A simple graph-based intermediate representation*. SIGPLAN Not., 30(3):35–49, 1993.

[CV98] Christiansen, Morten Voetmann und Per Velschow: *Region-based Memory Management in Java*. Diplomarbeit, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, May 1998.

[CWZ90] Chase, David R., Mark Wegman und F. Kenneth Zadeck: *Analysis of Pointers and Structures*. In: *Proc. of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*. ACM, June 20-22 1990.

[CX03] Cooper, Keith D. und Li Xu: *An Efficient Static Analysis Algorithm to Detect Redundant Memory Operations*. In: *Proc. of The Workshop on Memory System Performance (ISMM 2002)*, Band 38 der Reihe *SIGPLAN Not.*, Seiten 97–107. ACM, February 2003. Fortsetzung der Wertnummerierung auf Haldenobjekte.

[Det04] Detlefs, David: *A Hard Look at Hard Real-Time Garbage Collection*. Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), 1:23–32, 2004.

[DKAL05] Dhurjati, Dinakar, Sumant Kowshik, Vikram Adve und Chris Lattnerooper: *Memory Safety without Garbage Collection for Embedded Applications*. Trans. on Embedded Computing Sys., 4(1):73–111, 2005.

[DMM98] Diwan, Amer, Kathryn S. McKinley und J. Eliot B. Moss: *Type-Based Alias Analysis*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montral, Quebec, June 1998.

[Dol97] Dolby, Julian: *Automatic Inline Allocation of Objects*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997. ACM.

[ES90] Ellis, Margaret A. und Bjarne Stroustrop: *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[GA98] Gay, David und Alex Aiken: *Memory Management with Explicit Regions*. In: *Conference on Programming Languages, Design and Implementation*. EECS Department, University of California, Berkley, 1998.

[GA01] Gay, David und Alex Aiken: *Language Support for Regions*. In: *Conference on Programming Languages, Design and Implementation*. EECS Department, University of California, Berkley, 2001.

[GE90] Grosch, Josef und Helmut Emmelmann: *A tool box for Compiler Construction*. In: *LNCS 477: Compiler compilers; Third International Workshop, CC'90*, Schwerin, Germany, October 1990. Springer-Verlag, Berlin.

[GJS97] Gosling, J., B. Joy und G. Steele: *The Java Language Specification*. Addison-Wesley, 1997.

[Goo96] Goos, Gerhard: *Sather-K - The Language*. Technischer Bericht, Universität Karlsruhe (TH) - Fakultät für Informatik, 1996.

[GS00a] Gay, David und Bjarne Steensgaard: *Fast Escape Analysis and Stack Allocation for Object-Based Programs*. In: *Conference on Programming Languages, Design and Implementation; Compiler Construction*, 2000.

[GS00b] Gay, David und Bjarne Steensgaard: *Stack Allocating Objects in Java*. Technischer Bericht, Microsoft Resarch, 2000.

[Hac06] Hack, Sebastian: *Register Allocation for Programs in SSA-Form*. Doktorarbeit, Universität Karlsruhe (TH), 2006.

[Hal99] Hallenberg, Niels: *Combining Garbage Collection and Region Inference in The ML Kit*. Diplomarbeit, Department of Computer Science, University of Copenhagen, June 1999.

[Han90] Hanson, David R.: *Fast Allocation and Deallocation of Memory Based on Object Life times*. Software — Practice and Experience, 20(1):5–12, January 1990.

[HDH04] Hirzel, Martin, Amer Diwan und Michael Hind: *Pointer analysis in the presence of dynamic class loading*. In: *European Conference for Object-Oriented Programming (ECOOP)*, 2004.

[HHDH02] Hirzel, Martin, Johannes Henkel, Amer Diwan und Michael Hind: *Understanding the connectivity of heap objects*. In: *International Symposium on Memory Management (ISMM)*, 2002.

[HHN94] Hummel, Joseph, Laurie J. Hendren und Alexandru Nicolau: *A Language for Conveying the Aliasing Properties of Dynamic, Pointer-Based Data Structures*. In: *Proc. or the 8th International Parallel Processing Symposium*, Seiten 208–216, Cancun, Mexico, 1994. IEEE.

[Hin01] Hind, Michael: *Pointer analysis: Haven't we solved this problem yet?* In: *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Seiten 54–61, New York, NY, USA, 2001. ACM Press. Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.

[Hir04] Hirzel, Martin: *Connectivity-Based Garbage Collection*. Doktorarbeit, University of Colorado, 2004.

[HN90] Hendren, Laurie J. und Alexandru Nicolau: *Parallelizing Programs with Recursive Data Structures*. Technischer Bericht, University of California, Irvine, 1990.

[HP01] Hennessy, J. L. und D. A. Patterson: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishing Co., Menlo Park, CA., 3rd Auflage, 2001.

[HPR89] Horwitz, Susan, Phil Pfeiffer und Thomas Reps: *Dependence Analysis for Pointer Variables*. In: *Proc. of the SIGPLAN'89 CoPLDaI*, 1989.

[HS89] Hill, Mark und Alan Jay Smith: *Evaluating Associativity in CPU Caches*. In: *IEEE Trans. on Computers*, Band 38 der Reihe *IEEE Trans. on Computers*, December 1989.

[JKSZ05] Joukov, N., A. Kashyap, G. Sivathanu und E. Zadok: *Kefence: An Electric Fence for Kernel Buffers*. In: *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability (StorageSS 2005), "The Paradigm Shift to Info-Centric Protection," held in conjunction with the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, Seiten 37–43, FairFax, VA, November 2005. ACM.

[Joh97] Johnstone, Mark S.: *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. Doktorarbeit, The University of Texas at Austin, December 1997.

[Jon99] Jones, Richard: *Garbage Collection*. John Wiley & Sons, Ltd, 1999.

[Jon00] Jones, Simon Peyton: *Tackling the Awkward Squad: Monadic input/output, Concurrency, Exceptions, and foreign-language calls in Haskell*. In: *2000 Marktoberdorf Summer School*, 2000.

[JW93] Jones, Simon L Peyton und Philip Wadler: *Imperative functional programming*. In: *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Seiten 71–84, New York, NY, USA, 1993. ACM Press.

[Kam98a] Kamp, Poul-Henning: *Malloc(3) Revisited*. In: *USENIX 1998 Annual Technical Conference*. USENIX, the Advanced Computing Systems Association, June 1998.

[Kam98b] Kamp, Poul-Henning: *Malloc(3) Revisited*. FreeBSD System Documentation, The FreeBSD Project, 1998.

[KCL+99] Kennedy, R., S. Chan, S.-M. Liu, R. Lo, P. Tu und F. Chow: *Partial redundancy elimination in SSA form*. ACM Trans. on Programming Languages and Systems, 21(3):627–676, 1999.

[Kem95] Kempe, Magnus: *Heterogeneous Data Structures and Cross-Classification of Objects with Ada 95*. In: *Ada-Europe '95 Conference Proceedings*. Springer-Verlag, 1995.

[Knu73] Knuth, Donald E.: *The Art of Computer Programming*, Band 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973. First edition published in 1968.

[Koe97] Koenig, Andrew: *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, Band WG21/N1043 der Reihe *Project:Programming Language C++*. AT&T Research, AT&T Research, PO Box 636, 600 Mountain Avenue, Room 2C-306, Murray Hill, NJ 07974 USA, 1997.

[KP96] Kiem-Phong, Vo.: *Vmalloc: A General and Efficient Memory Allocator*. In: *Software — Practice and Experience*, Band 26, Seiten 357–374, March 1996.

[Lea96] Lea, Doug: *Implementations of malloc*. Available at http://g.oswego.edu/dl/html/malloc.html, 1996.

[LH83] Lieberman, Henry und Carl Hewitt: *A Real Time Garbage Collector Based On The Lifetimes of Objects*. ACM Transactions on Programming Languages and Systems, 26(6):419–429, June 1983.

[Lie06] Liekweg, Florian: *Compiler-directed, Automatic Memory Management*. In: *SPACE 2006 — Third workshop on Semantics, Program Analysis, and Computing Environments for Memory Management. Co-located with POPL 2006.* ACM Press., Charleston, SC., 2006.

[LML+93] Lumetta, Steve, Liam Murphy, Xiaoye Li, David C. Culler und Ismail S. Khalil: *Decentralized Optimal Power Pricing: The Development of a Parallel Program*. IEEE Parallel Distrib. Technol., 1(4):23–31, 1993.

[LR92] Landi, William A. und Barbara G. Ryder: *A safe Approximation Algorithm for interprocedrual Pointer Aliasing*. In: *[PLD92]*, Seiten 235–248, 1992.

[McB64] McBeth, J. Harold: *On the reference counter method*. Comm. of the ACM, 6(9):575, 1964.

[Mey92] Meyer, Bertrand: *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mor98] Morgan, C. Robert: *Building an Optimizing Compiler*. Digital Press, Februar 1998. ISBN 155558179X.

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Juli 1997. ISBN 1558603204.

[Mye96] Myers, Scott: *Effective C++*. Addison-Wesley, 1996.

[Mye97] Myers, Scott: *More Effective C++*. Addison-Wesley, 1997.

[Nil93] Nilsen, Kelvin D.: *Reliable real-time garbage collection of C++*. In: Moss, Eliot, Paul R. Wilson, und Benjamin Zorn (Herausgeber): *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, Band 3. ACM, October 1993.

[NKY04] Nagata, Akihito, Naoki Kobayashi und Akinori Yonezawa: *Region-Based Memory Management for a Dynamically-Typed Language*. In: Schmidt, David (Herausgeber): *Programming Languages and Systems*, Band 2986 der Reihe *Lecture Notes in Computer Science*. Dept. of Computer Science, Graduate School of Information Science and Technology, University of Tokyo, Springer-Verlag, Berlin, 2004.

[NNH99] Nielson, Flemming, Hanne R. Nielson und Chris Hankin: *Principles of Program Analysis*. Springer-Verlag, Berlin, November 1999. ISBN 3540654100.

[Piz99] Pizzi, Cesare: *Memory Access Error Checkers*. Linux J., 1999(61es):26, 1999.

[PL94] Pande, Hermant D. und William A. Landi: *Interprocedrual Def-Use Associations for C systems with Single Level Pointers*. In: *IEEE-TSE 1994*, 1994.

[PLB+00] Probst, Christian, Götz Lindenmaier, Marcel Beemster, Tobias Ritzau und Florian Liekweg: *Joc — The Joses Compiler*. Technischer Bericht, Universität des Saarlands; IPD, Universität Karlsruhe (TH); ACE bv, 2000.

[QH02] Qian, Feng und Laurie Hendren: *An Adaptive, Region-based Allocator for Java*. In: *Proceedings of the third International Symposium on Memory Management*, Seiten 127 – 138. ACM Press, 2002.

[Ram94] Ramalingam, G.: *The undecidability of aliasing*. ACM Trans. Program. Lang. Syst., 16(5):1467–1471, 1994.

[RBPL00] Ritzau, Tobias, Marcel Beemster, Christian W. Probst und Florian Liekweg: *JoC - the JOSES Compiler*. In: *Java for Embedded Systems Workshop*, 2000.

[Rit03] Ritzau, Tobias: *Memory Efficient Hard Real-Time Garbage Collection*. Doktorarbeit, Linköping University, Mai 2003.

[RVW+97] Romer, Ted, Geoff Voelker, Alec Wolman, Dennis Lee, Hank Levy, Wayne Wong, Brian Bershad und Brad Chen: *Instrumentation and Optimization of Win32/Intel Executables Using Etch*. In: *Proceedings of the Usenix Windows NT Workshop*, 1997.

[RW77] Roth, David J. und David S. Wise: *The One-Bit Reference Count*. BIT, 17:351–359, 1977.

[RW98] Roth, David J. und David S. Wise: *One-bit counts between unique and sticky*. In: *ISMM '98: Proceedings of the 1st international symposium on Memory management*, Seiten 49–56, New York, NY, USA, 1998. ACM Press.

[Sie98] Siebert, Fridtjof: *Guaranteeing Non-Disruptiveness and Real-Time Deadlines in an Incremental Garbage Collector*. In: *International Symposium on Memory Management 1998 (ISMM'98)*, Band 34 der Reihe *ACM SIGPLAN Notices*, 1998.

[Sie99] Siebert, Fridtjof: *Hard Real-Time Garbage-Collection in the Jamaica Virtual Machine*. In: *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, Seite 96, Washington, DC, USA, 1999. IEEE Computer Society.

[Spo04] Spolsky, Joel: *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress, August 2004.

[SRW96] Sagiv, M., T. Reps und R. Wilhelm: *Solving Shape-Analysis Problems in Languages with Destructive Updating*. In: *Conference Record of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[Ste95] Steensgaard, Bjarne: *Sparse functional stores for imperative programs*. In: *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, Seiten 62–70, New York, NY, USA, 1995. ACM Press.

[Str91] Stransky, Jan: *Semantic Analysis of dynamically allocated Data*. In: *Workshop Compilers for Parallel Computers*, 1991.

[TB97] Tofte, Mads und Lars Birkedal: *Region-based memory management*. Information and Computation, 132(2):109–176, February 1997.

[TB98] Tofte, Mads und Lars Birkedal: *A Region Inference Algorithm*. ACM Transactions on Programming Languages and Systems, 20(5):724–767, July 1998.

[TLB99] Trapp, Martin, Götz Lindenmaier und Boris Boesler: *Documentation of the Intermediate Representation FIRM*. Technischer Bericht 1999-44, Universität Karlsruhe (TH), Dez 1999.

[Tra00] Trapp, Martin: *Optimierung Objektorientierter Programme*. Doktorarbeit, Universität Karlsruhe (TH), Fakultät für Informatik, Institut für Programmstrukturen und Datenorganisation, 2000.

[TT93] Tofte, Mads und Jean-Pierre Talpin: *A Theory of Stack Allocation in Polymorphically Typed Languages*. Technischer Bericht 93/15, Department of Computer Science, Copenhagen University, June 1993.

[TT94] Tofte, Mads und Jean-Pierre Talpin: *Implementation of the typed call-by-value λ-calculus using a stack of regions*. In: *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Seiten 188–201, New York, NY, USA, 1994. ACM Press.

[WG84] W.M.Waite und G.Goos: *Compiler Construction*. Springer-Verlag, Berlin, 1984.

[Wil90] Wilson, Paul R.: *Uniprocessor Garbage Collection Techniques*. ACM Computing Surveys, 637:1 – 34, September 1990.

[Wir05] Wirth, Niklaus: *Compiler Construction*. Nummer ISBN 0-201-40353-6. Addison-Wesley, 2005. This is a slightly revised version of the book published by Addison-Wesley in 1996.

[WJ93] Wilson, Paul R. und Mark S. Johnstone: *Real-Time Non-Copying Garbage Collection*. In: *1993 ACM OOPSLA Workshop on Memory Management and Garbage Collection*, 1993. Position paper.

[WJNB95a] Wilson, Paul R., Mark S. Johnstone, Michael Neely und David Boles: *Dynamic Storage Allocation: A Survey and Critical Review*. In: *IWMM '95: Proceedings of the International Workshop on Memory Management*, London, UK, 1995. Springer-Verlag, Berlin.

[WJNB95b] Wilson, Paul R., Mark S. Johnstone, Michael Neely und David Boles: *Memory allocation policies reconsidered*. Technischer Bericht, University of Texas at Austin, Department of Computer Sciences, London, UK, 1995.

[WL02] Whaley, John und Monica S. Lam: *An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages*. In: *SAS2*, Seiten 180–195, London, UK, 2002. Springer-Verlag, Berlin.

[WM96] Willhelm, Reinhard und Dieter Maurer: *Übersetzerbau. Theorie, Konstruktion, Generierung*. Springer-Verlag, Berlin, 1996. ISBN 3540616926.

[WSB01] William S. Beebee, Jr.: *Region-Based Memory Management for Real-Time Java*. Doktorarbeit, Massachusetts Institute of Technology, September 2001.

„The first implementation of this algorithm was actually a file system, done in assembler using 5-hole "Baudot" paper tape for a drum storage device attached to a 20 bit germanium transistor computer with 2000 words of memory, but that was many years ago."

Poul-Henning Kamp, [Kam98a]