Studienarbeit

# Ein neuer Speicher-Allokator für XEmacs

Marcus Crestani
marcus@crestani.de

Betreuer:
Prof. Dr. Herbert Klaeren
Dr. Michael Sperber

April 2004

Arbeitsbereich Programmiersprachen und Übersetzer
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

**Zusammenfassung**

Die automatische Speicherverwaltung des XEmacs ist langsam: der Benutzer wird in seinem Arbeiten durch lange Programmpausen unterbrochen. Zudem ist der Speichermanager sehr eng mit dem restlichen Programmcode verwoben, es gibt keine übersichtliche Schnittstelle zwischen XEmacs und Speicherverwaltung.

In dieser Arbeit beschreibe ich einen neuen Allokator für XEmacs, einen wichtigen Teil der automatischen Speicherverwaltung. Mein Ziel ist es, den Speichermanager zu beschleuningen, ihn vom Rest des XEmacs zu trennen und die Schnittstelle zum Speichermanager zu verkleinern.

Ich stelle den aktuellen Speichermanager in dieser Arbeit im Detail vor. Davon ausgehend beschreibe ich die vorgenommenen Änderungen in den Bereichen Datenstrukturen, Allokation und Speicherrückgewinnung.

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

## 1.1 Was ist XEmacs?

XEmacs ist ein programmierbarer Editor, dessen Fähigkeiten über einfaches Einfügen und Entfernen hinausgehen. Dazu gehört automatische Einrückung und Einfärben von Programmtexten verschiedener Programmiersprachen und das Anzeigen mehrerer Dateien gleichzeitig.

Da XEmacs programmierbar ist, ist er beliebig erweiterbar. XEmacs enthält die Implementierung einer eigenen Programmiersprache namens *Emacs Lisp* (auch *ELisp* genannt). Ein Großteil von XEmacs und alle seine Erweiterungen sind in diesem Lisp-Dialekt geschrieben.

Wie jede Lisp-artige Programmiersprache setzt auch Emacs Lisp eine automatische Speicherverwaltung voraus. Die grundlegende Funktionalität des XEmacs ist in der Programmiersprache C geschrieben, dazu gehört die Implementierung der Speicherverwaltung von Emacs Lisp.

## 1.2 Motivation der Arbeit

XEmacs ist ein über Jahre gewachsenes Programm; seine automatische Speicherverwaltung stammt aus den siebziger Jahren.

Ein Benutzer des XEmacs wird während des Arbeitens mit den Folgen der veralteten Speicherverwaltung konfrontiert: oft scheint das System zu stocken, die Anzeige friert ein und das Programm zeigt keinerlei Reaktion auf Tastatureingaben. Schuld daran ist die automatische Speicherverwaltung, die unzumutbar lange Pausen verursacht. Sie ist zu langsam.

Auch für den Programmierer wird die Arbeit erschwert: Der Programmcode ist in weiten Teilen sehr unübersichtlich. Der Speichermanager ist sehr eng mit dem restlichen Programmcode verwoben; einfache Änderungen an XEmacs können zu weitgehenden Änderungen im Speichermanager führen. Der Entwickler muss sich daher um viele Details kümmern.

Ziel dieser Studienarbeit ist es, einen wichtigen Teil der automatischen Speicherverwaltung, den Allokator, zu ersetzen. Damit wird eine Grundlage geschaffen, die es ermöglichen soll, eine schnellere Speicherverwaltung in XEmacs zu integrieren.

In Kapitel 2 gebe ich einen Überblick über das Gebiet der automatischen Speicherverwaltung. In Kapitel 3 beschreibe ich die aktuelle Situation in XEmacs mit einer Bewertung und der Vorstellung notwendiger Änderungen. Den neuen Allokator stelle ich in Kapitel 4 vor, die Änderungen am XEmacs-Programmcode in Kapitel 5. Ein Fazit und Informationen über zukünftige Vorhaben sind in Kapitel 6 zu finden.

# Kapitel 2

# Automatische Speicherverwaltung

In diesem Kapitel gebe ich eine Einführung in das Gebiet der Speicherverwaltung. Im Folgenden stelle ich die drei Methoden der Speicherverwaltung vor: *statische Allokation*, *Stack-Allokation* und *Allokation auf dem Heap*.

**Statische Allokation**
: Die einfachste Art, Speicher zu verwalten, ist die statische Allokation. Der ganze benötigte Speicher ist bereits während der Übersetzung bekannt; zur Laufzeit wird nur einmal die vorher bekannte Menge Speicher angefordert. Diese Vorgaben sind nicht für alle Programme ausreichend: Programme, deren Speicherbedarf während der Laufzeit in vorher unbekanntem Umfang steigt, können so nicht funktionieren. Möglich werden solche Programme mit den Allokationsstrategien, die ich im Folgenden vorstelle.

**Stack-Allokation**
: Bei der Stack-Allokation werden Daten auf dem Laufzeit-Stack gespeichert. Bei jedem Prozeduraufruf wird dem Stack ein *Aktivierungsblock* angelegt, der lokale Datenstrukturen enthält. Beim Verlassen der Prozedur wird der Aktivierungsblock wieder entfernt. Auf dem Stack allozierte Daten leben nur so lange, wie ihr Aktivierungsblock, also bis zum Verlassen der Prozedur.

**Allokation auf dem Heap**
: Die flexibelste Methode, Datenstrukturen Speicher zuzuweisen, ist die Allokation auf dem Heap. Der Heap ist ein Bereich im Speicher, auf dem Datenstrukturen beliebig angelegt werden können. In der Programmiersprache C kann das Programm Speicher auf dem Heap mit der Funktion `malloc` belegen. Das Objekt lebt, solange das Programm läuft oder es explizit mit `free` freigegeben wird. Die Freigabe des Speichers von Hand heißt *manuelle Speicherverwaltung*.

Ein Programmierer muss mit der manuellen Speicherverwaltung sehr sorgfältig sein. Wird Speicher freigegeben, der vom Programm aus noch benutzt wird, wurde dieser Speicher zu früh freigegeben. Zu früh freigegebener Speicher kann zu Programmabstürzen und Fehlverhalten des Programms führen, da auf Speicher

zugegriffen wird, der gar nicht mehr dem Programm zugeteilt ist. Wird Speicher, der nicht mehr gebraucht wird, zu spät oder gar nicht freigegeben, entsteht ein *Speicherleck*. Dies kann dazu führen, dass der Hauptspeicher voll belegt ist, obwohl die meisten Objekte gar nicht mehr benutzt werden. Wird zum Beispiel die letzte Referenz auf eine Datenstruktur gelöscht, ohne den Speicher dieser Datenstruktur vorher freizugeben, kann dieser Speicher im laufenden Programm nicht mehr freigegeben werden.

Die *automatische Speicherverwaltung* übernimmt die Allokation und die Speicherfreigabe von Heap-Objekten. Das Programm fordert Speicher an, der Programmierer muss aber nicht mehr auf die Speicherfreigabe achten. Ist reservierter Speicher vom Programm nachweislich nicht mehr erreichbar, wird er von der Speicherverwaltung freigegeben.

## 2.1 Vor- und Nachteile

Der automatischen Speicherverwaltung wird nachgesagt, Programme deutlich zu verlangsamen und interaktive Systeme zu unterbrechen. Der aktuelle XEmacs bestärkt diese Meinung.

Jones und Lins [JL96] erwähnen Messungen, in denen Programme aus den Siebzigern bis zu 40% ihrer Laufzeit für die automatische Speicherverwaltung aufbrachten. Dank neuer Techniken kann sich dieser Anteil auf 10% der Laufzeit verringern — dadurch wurde der Hauptkritikpunkt der automatischen Speicherverwaltung entwertet.

Vor allem für den Programmierer bringt eine automatische Speicherverwaltung viele Vorteile. Der Programmieraufwand wird reduziert, da er sich nicht um Speicherverwaltung kümmern muss. Dadurch wird das Programm übersichtlicher und korrekter. Heutzutage überwiegen die Argumente für automatische Speicherverwaltung deutlich.

Daher ist es nicht verwunderlich, dass neuere, kommerziell orientierte Programmiersprachen wie *Java* und *C#* und andere wie *Scheme*, *ML*, *Haskell*, *Prolog*, *Smalltalk* und eben die in XEmacs implementierte Programmiersprache *Emacs Lisp* eine automatische Speicherverwaltung vorsehen.

## 2.2 Überblick und Terminologie

An dieser Stelle definiere ich die wichtigsten Begriffe des Gebiets der automatischen Speicherverwaltung. Im Deutschen ist auch der englische Begriff *Garbage Collection*, kurz *GC*, gebräuchlich.

Ein Programm mit automatischer Speicherverwaltung, setzt sich aus dem *Speichermanager* oder *Kollektor* und dem Teil des Programms zusammen, das die „nützliche" Arbeit macht. Dieser Teil wird in der Literatur *Mutator* oder *Klient* genannt.

Der Speichermanager stellt auf Anfrage des Klienten Speicher zur Verfügung und ermittelt automatisch, wann dieser Speicher für den Klienten nicht mehr erreichbar ist. Erreichbarer Speicher heißt *lebendiger Speicher*, nicht mehr erreichbaren Speicher ist *toter Speicher*. Der Speichermanager entzieht toten Objekten Speicher.

Um festzustellen, welche Objekte noch am Leben sind, stellt der Speichermanager einen *Erreichbarkeitsgraphen* auf. Als Ausgangsbasis dient das *Root-Set*. Das Root-Set wird dem Speichermanager vom Klienten bekannt gemacht. Von dort aus sind die lebendigen Objekte über Zeiger miteinander verbunden. Alle Objekte, die über lebendige Zeiger erreichbar sind, werden in den Erreichbarkeitsgraphen aufgenommen. Der Mutator verändert den Graphen der lebendigen Objekte.

Innerhalb des Speichermanagers gibt es ein Modul mit dem Namen *Allokator*. Der Allokator verwaltet freien Speicher und bedient die Speicheranforderungen des Klienten. Die Verwaltung des freien Speichers ist kompliziert, da Kollektoren mit der Zeit den Speicher fragmentieren: Durch die Auflösung von Objekten kommt es zu Lücken im allozierten Speicher. Um zusammenhängenden Speicher einer bestimmten Größe schnell finden zu können, benutzt der Allokator dazu eine Datenstruktur namens *Freiliste*. Genau genommen handelt es sich um mehrere Datenstrukturen, die jeweils unbenutzten Speicherbereich einer Größe als verkettete Liste darstellen. Wird neuer Speicher angefordert, werden die Freilisten traversiert, um ein Stück passenden Speicher zu finden.

Einen Allokator werde ich im Rahmen dieser Arbeit implementieren.

# Kapitel 3

# Automatische Speicherverwaltung in XEmacs

In diesem Kapitel werde ich die existierende automatische Speicherverwaltung des XEmacs vorstellen. Nach der Bewertung der Speicherverwaltung am Ende des Kapitels gehe ich auf die notwendigen Änderungen ein.

Zusätzlich beschreibe ich den *portablen Dumper*, da meine Änderungen am Allokator auch eine Anpassung des portablen Dumpers nach sich ziehen. Der portable Dumper kann das Speicherabbild des laufenden XEmacs in eine Datei schreiben und aus der Datei wiederherstellen. Verwendet wird der portable Dumper, um die Ladezeit des XEmacs zu verkürzen.

## 3.1 Allokation in XEmacs

Im XEmacs gibt es verschiedene Strategien, Lisp-Objekte auf dem Heap anzulegen. Grundsätzlich werden Lisp-Objekte in drei große Bereiche eingeteilt, die unterschiedlich alloziert werden: einfache Datentypen, Lrecords und Lcrecords [WBN+]. Zu den *einfachen Datentypen* zählen nur *Integers* (ganze Zahlen) und *Chars* (Zeichen). Die größeren Objekte, *Lrecords* und *Lcrecords* in XEmacs-Terminologie, werden auf dem Heap gespeichert und jeweils durch einen Zeiger repräsentiert. Für zwei XEmacs-Datentypen gibt es zusätzliche Vorgehensweisen, Heapspeicher zu allozieren: *Strings* (Zeichenketten) und *Buffer*. Im Folgenden werde ich die unterschiedlichen Allokationsstrategien vorstellen.

### 3.1.1 Einfache Datentypen

Integer und Chars werden nicht auf dem Heap abgelegt, sondern passen in ein 32-Bit- bzw. 64-Bit-Maschinenwort. Zusammen mit einer Typ-Information bilden solche einfachen Werte *Deskriptoren*.

Größere Objekte werden auf dem Heap gespeichert und durch einen Zeiger repräsentiert, der auch als Deskriptor dargestellt wird.

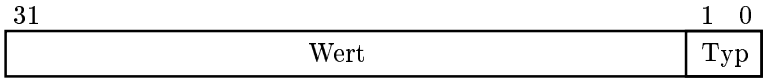


Abb. 3.1: Bit-Belegung eines Deskriptors.[Rei01]

| Typ | Bits 31-2 | Bits 1-0 |
|---|---|---|
| Heap-Objekt | Zeiger | 00 |
| gerade ganze Zahl | Wert / 2 | 01 |
| Zeichen | Wert | 10 |
| ungerade ganze Zahl | (Wert - 1) / 2 | 11 |

Tabelle 3.1: Repräsentation der Lisp-Objekte durch Deskriptoren.[Rei01]

Die Unterscheidung von geraden und ungeraden ganzen Zahlen führt zu einer effektiven Nutzung von 31 Bits für die Darstellung einer ganzen Zahl.

### 3.1.2 Lrecords

Zu den Lrecords zählen in erster Linie Objekte, die sehr häufig vorkommen, relativ klein sind und innerhalb eines Typs die gleiche Größe haben. Dazu zählen die XEmacs-Datentypen *cons* (Paare), *subr* (aus Lisp aufrufbare C-Funktion), *float* (Gleitkommazahl), *compiled function* (Byte-Code), *symbol* (Symbol), *extent* (Bereich in einem Buffer), *event* (Ereignis) *marker* (Position in einem Buffer) und *string* (Zeichenketten), deren Besonderheit im Weiteren beschrieben wird.

Ein Lrecord besteht aus einem Kopf und einem Datenteil. Der Kopf enthält eine Typbezeichnung, ein Mark-Bit und Schreibschutz-Bits.

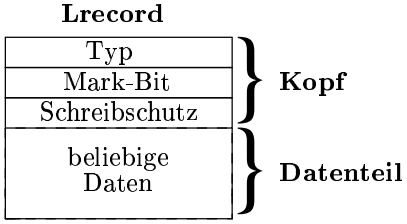


Abb. 3.2: Aufbau eines Lrecords.

Ein Beispiel für einen Lrecord ist `cons`:

```
struct Lisp_Cons
{
  struct lrecord_header lheader;
  Lisp_Object car_, cdr_;
};
```

Das Feld `lheader` enthält die in Abbildung 3.2 dargestellten Typ- und Verwaltungsinformationen. Im Datenteil des Lrecords gibt es die Felder `car_` und `cdr_`, die jeweils einen Zeiger-Deskriptor enthalten.

Alle Lrecords eines bestimmten Typs haben die gleiche Größe. Sie werden durch sogenannte *FROB-Blöcke* verwaltet. Ein FROB-Block ist ein Stück zusammenhängender Speicher. Für jeden Typ gibt es FROB-Blöcke, unterteilt in Zellen. In eine solche Zelle passt genau ein Lrecord des entsprechenden Objekttyps. Die FROB-Blöcke eines Lrecord-Typs sind doppelt verkettet; sie bilden die Freiliste für einen Lrecord-Typ.

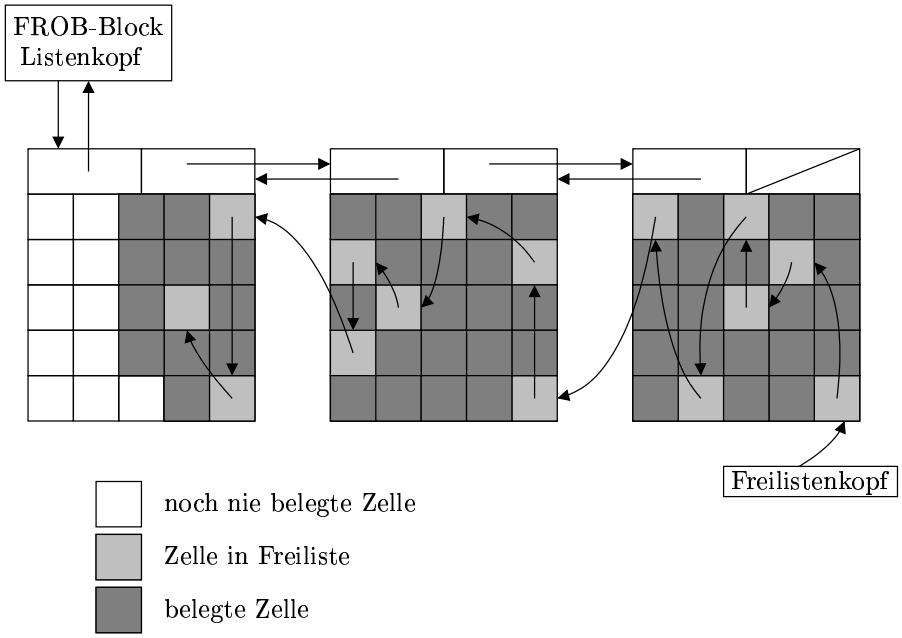


Abb. 3.3: Speicherorganisation mit FROB-Blocks.[Rei01]

Lrecords werden aufgrund ihrer festen Größe auf einfache Weise mit Hilfe von Freilisten alloziert und verwaltet.

### 3.1.3 Lcrecords

Der Aufbau eines Lcrecords ist ähnlich zu dem eines Lrecords, allerdings haben die Objekte eines Typs einen unterschiedlich großen Datenteil. Lcrecords können nicht mit FROB-Blöcken verwaltet werden, da ihre Größe variabel ist.

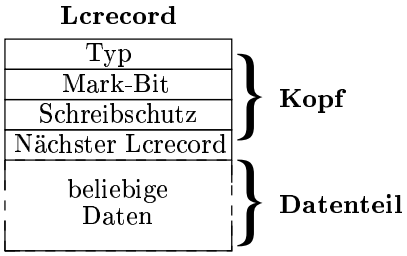


Abb. 3.4: Aufbau eines Lcrecords.

Ein typisches Beispiel für einen Lcrecord ist das Objekt `Vector`. Die Größe eines Vektors hängt vom Feld `contents` ab:

```
struct Lisp_Vector
{
  struct lcrecord_header header;
  long size;
  Lisp_Object contents[1];
};
```

Der Kopfteil eines Lcrecords enthält zusätzlich zu den Verwaltungsinformationen, die auch ein Lrecord besitzt, einen Zeiger auf den nächsten Lcrecord. Diese Kopfinformationen sind im Feld `header` gespeichert. Ein Vektor enthält die Anzahl der gespeicherten Einträge im Feld `size`, die Einträge selbst werden im Array `contents` gespeichert. Das Feld `contents` ist als *dehnbares Array* mit nur einem Element deklariert. Da das Vektor-Objekt jedoch mit der richtigen Größe im Speicher alloziert wird, kann auf alle Elemente zugegriffen werden.

Lcrecords werden einzeln mit `malloc` im Speicher angelegt und sind über das Feld „nächster Lcrecord" in einer Liste verkettet. Freigegeben werden sie mit `free`.

Daher gibt es für Lcrecords keine eigene Freilistenverwaltung. Die Speicherorganisation wird komplett der Speicherverwaltung des Betriebssystems überlassen.

### 3.1.4 Strings

Die Besonderheit bei Zeichenketten ist, dass sie zweigeteilt im Speicher abgelegt werden. Der Datensatz, der Informationen über die Zeichenkette enthält, wird als Lrecord in einem FROB-Block alloziert; die Daten, der eigentliche Text, wird gesondert abgespeichert. Wegen der variierenden Länge des Datenteils wird nicht der komplette String in einem FROB-Block abgespeichert, da mit FROB-Blöcken ausschließlich Objekte fester Größe verwaltet werden.

```
struct Lisp_String
{
  struct lrecord_header lheader;
  Bytecount size_;
  Ibyte *data_;
  Lisp_Object plist;
};
```

Das Feld `data_` verweist auf den Datenteil der Zeichenkette, die in einer Struktur namens `strings_chars_block` abgespeichert ist. Diese Struktur speichert mehrere kurze Zeichenketten hintereinander. Wird eine Zeichenkette freigegeben, schließt der Speichermanager die entstandene Lücke durch zusammenschieben der übrigen Zeichenketten. Dadurch wird eine Fragmentierung des Speichers verhindert.

Dies wird in XEmacs allerdings nur für kurze Zeichenketten angewendet. Längere Zeichenketten werden direkt mit dem Betriebssystem-`malloc` alloziert.

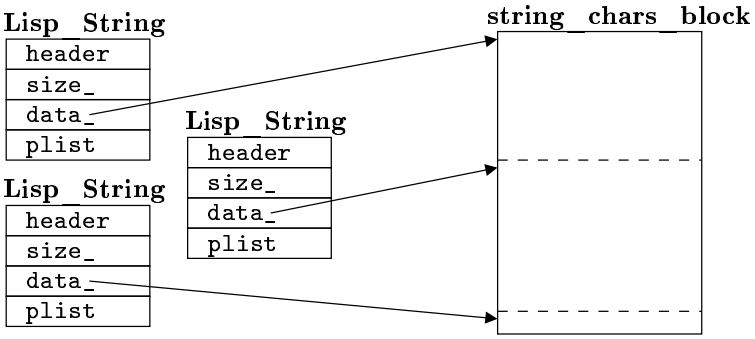


Abb. 3.5: Allokation von Zeichenketten.

### 3.1.5 Buffer

Ein Buffer ist ein Objekt, das Text enthält, der angezeigt und editiert werden kann. Er ist ähnlich zu einem String, allerdings ist ein Buffer optimiert für das Einfügen und Löschen von Textteilen. Wie ein String wird auch ein Buffer zweigeteilt im Speicher angelegt. Da es ein großer Datensatz ist, wird die Information über einen Buffer als Lcrecord alloziert. Für den Buffertext, der sehr groß werden kann, wird separat Speicher angefordert.

Grundsätzlich wird kein Speicher des Bufferinhalts freigegeben, solange ein Buffer existiert. Wird eine 20-Megabyte-Datei mit XEmacs geöffnet und der Inhalt bis auf ein Zeichen gelöscht, werden trotzdem immer noch die vollen 20 Megabyte belegt. Erst wenn der Buffer geschlossen wird (in XEmacs-Terminologie heißt das *den Buffer killen*), wird dieser Speicherbereich freigegeben.

Bei `malloc`-Implementierungen, die keinen Speicher an das Betriebssystem zurückgeben, bleibt der Speicher dem Prozess zugeordnet, es werden unnötig Ressourcen verschwendet. Im XEmacs-Programmcode findet sich dafür eine Optimierung: der *Relocating Allocator* stellt sicher, dass freigegebener Speicher an das Betriebssystem zurückgegeben wird. Der Relocating Allocator benutzt für die Belegung großer Bereiche `mmap`. Die Freigabe mit `munmap` garantiert, dass der Speicher an das Betriebssystem zurückgegeben wird.

Viele Implementierung von `malloc` benutzen `mmap`, um große Speicherbereiche anzufordern. Nur wenn auf dem System `mmap` zur Verfügung steht, es aber nicht von `malloc` benutzt wird, wird der „selbstgemachte" Relocating Allocator zum Allozieren der Buffertexte benutzt. Das ist inzwischen selten der Fall, in den meisten Fällen wird das normale Betriebssystem-`malloc` verwendet.

Die Eigenschaft, die dem *Relocating Allocator* den Namen gibt, ist die *Defragmentierung des Speichers* durch selbstständiges Verschieben von Speicherblöcken. Nach Wing et al. [WBN+] führt dies aber zu einem bemerkbaren Geschwindigkeitsnachteil.

## 3.2 Garbage Collection

XEmacs benutzt das *Mark-Sweep-Verfahren*, benannt nach seinen beiden Phasen, der Markierungs- und Sweep-Phase. Es ist ein einfaches, nicht-bewegendes Verfahren. Das heißt, die Objekte bleiben, solange sie leben, an der gleichen Speicherstelle.

Der Kollektor markiert in der Markierungsphase jedes Objekt, das erreichbar ist. Markierte Objekte gelten als lebendig. In der Sweep-Phase werden alle belegten FROB-Blöcke und Lcrecords vom Speichermanager besucht. Markierte Objekte bleiben an Ort und Stelle erhalten, der Kollektor löscht unmarkierte Objekte und gibt ihren Speicher frei.

### 3.2.1 Markierungsphase

Um festzustellen, welche Objekte noch am Leben sind, stellt der Kollektor einen Erreichbarkeitsgraphen auf und traversiert ihn. Die Aufstellung des Erreichbarkeitsgraphen beginnt im Root-Set. Der Klient macht dem Kollektor das Root-Set bekannt. Um den Erreichbarkeitsgraphen traversieren zu können, muss der Speichermanager die von den verschiedenen Objekten ausgehenden Zeiger kennen. Dazu besitzt jedes Lisp-Objekt eine Beschreibung seines Aufbaus.

Zum Beispiel besteht ein Cons-Paar aus zwei Zellen, die beide beliebige Lisp-Objekte enthalten können. Hier die Definition und die *Speicherbeschreibung* eines Cons-Paares:

```
struct Lisp_Cons
{
  struct lrecord_header lheader;
  Lisp_Object car_, cdr_;
};

static const struct memory_description cons_description[] = {
  { XD_LISP_OBJECT, offsetof (Lisp_Cons, car_) },
  { XD_LISP_OBJECT, offsetof (Lisp_Cons, cdr_) },
  { XD_END }
};
```

Die Zellen `car_` und `cdr_` des Paares werden in der Speicherbeschreibung durch die Konstante `XD_LISP_OBJECT` als Lisp-Objekte gekennzeichnet. Die genaue Position der einzelnen Felder wird durch den Aufruf der Funktion `offsetof` berechnet. Durch die Offset-Angaben kann der Speichermanager die Elemente finden. Insgesamt sind damit alle lebendigen Zeiger bekannt, der Erreichbarkeitsgraph kann präzise traversiert werden.

### 3.2.2 Sweep-Phase

In der Sweep-Phase läuft der Speichermanager den kompletten Heap ab. Stößt er dabei auf Objekte, deren Markierungs-Bit nicht gesetzt ist, gibt er ihren Speicher frei. Der freie Speicher wird dann je nach Art des Objekts in die verschiedenen Freilisten eingehängt (siehe Abschnitt 3.1).

## 3.3 Der portable Dumper

Die meisten Funktionen des XEmacs sind in Emacs Lisp geschrieben. Diese Funktionen bei jedem Programmstart zu laden und auszuführen, dauert sehr lange; für einen Editor ist eine lange Startzeit nicht akzeptabel.

Daher wird bei der Installation nach dem Übersetzen des Systems XEmacs gestartet und der ganze Lisp-Code geladen und ausgeführt. Am Ende der Initialisierung wird schließlich ein Speicherbild des Prozesses in eine ausführbare Datei geschrieben. Wird diese Datei dann ausgeführt, stellt sie das Speicherbild wieder her. Dies beschleunigt das Starten des Editors erheblich.

Das Sichern des Prozesszustands und das Erzeugen einer ausführbaren Datei ist sehr vom Betriebssystem abhängig. Daher wurde der *portable Dumper* entwickelt, der unabhängig vom Betriebssystem ein Speicherabbild schreiben und wiederherstellen kann.

Der portable Dumper basiert, wie der Markierungs-Algorithmus, auf der Traversierung des Erreichbarkeitsgraphen. Er benutzt ebenfalls das Root-Set und die Speicherbeschreibungen der Lisp-Objekte (siehe Abschnitt 3.2.1).

## 3.4 Bewertung und nötige Änderungen

Die unterschiedliche Handhabung der Lrecords und Lcrecords macht die Arbeit des Programmierers unnötig kompliziert. Die Allokation der Lcrecords durch das Betriebssystem-`malloc` ist nicht effizient, es führt zu einer starken Fragmentierung des Speichers. Der Nachteil der Lrecords ist, dass sie zu eng mit dem Speichermanager verwoben sind: Änderungen an einem Lrecord-Datentyp bedeutet eine Änderung des Speichermanagers. Weitere Spezialfälle, wie zum Beispiel die Objekte String und Buffer, entstehen durch die existierende, größenbasierte Allokationstrategie. Dies trägt noch mehr zur Verwirrung des Programmierers bei.

Die oben genannten Nachteile des bisherigen Allokators werde ich im Rahmen dieser Arbeit verbessern. Durch den neuen Allokator wird sich die Schnittstelle zum Speichermanager verkleinern. Gleichzeitig wird ein einfacheres, da einheitliches, Objektsystem entstehen.

Dafür werden die vorteilhaften Eigenschaften der Objekttypen beibehalten: ich übernehme die Unabhängigkeit der Lcrecords vom Speichermanager und benutze die Allokation der Lrecords als Anregung für den neuen Allokationsalgorithmus.

Eine Unterscheidung von Lrecords und Lcrecords und andere Spezialbehandlungen sind in Zukunft nicht mehr nötig.

In den folgenden Kapiteln beschreibe ich meine Änderungen im Detail.

# Kapitel 4

# Der neue Allokator

Als Vorbild für den neuen Allokator habe ich den Allokator des Boehm-Demers-Weiser-Kollektors[Boe04], kurz Boehm-Kollektor, ausgewählt. Der Allokator wird als sehr effizient angesehen [JL96]. Die grundlegenden Ideen und Algorithmen meines Allokators sind dem Allokator des Boehm-Kollektors nachempfunden.

## 4.1 Drei-Stufen-Allokation

Der Allokator fordert auf unterster Ebene große Stücke zusammenhängenden Speichers vom Betriebssystem an. Diese großen Blöcke heißen *Heap-Sektionen*. Der Allokator übernimmt die Verwaltung dieses Speichers. Unterteilt werden Heap-Sektionen wiederrum in *Seiten* beziehungsweise *Mehrfachseiten*. Eine Seite ist ein Stück zusammenhängenden Speicherbereichs, eine Mehrfachseite ist eine ganzzahlige Anzahl zusammenhängender Seiten. Die *Seitengröße* ist im Programmtext durch die Variable `PAGE_SIZE` konfigurierbar. Wie der Name sagt, legt sie die Größe einer Seite in Bytes fest. Eine Seite besteht aus einer oder mehreren *Zellen*. Jede Zelle ist ein Speicherort für ein Objekt.

Wenn ein Speicherobjekt größer als `PAGE_SIZE` ist, wird es auf einer Mehrfachseite alloziert. Die Zelle dieses Objekts erstreckt sich über die ganze Mehrfachseite; eine Mehrfachseite enthält immer nur ein Objekt. Ist ein Speicherobjekt kleiner als $\frac{1}{2}$ `PAGE_SIZE` , dann enthält eine Seite mehrere dieser Objekte. Je kleiner die Objekte, desto mehr Zellen passen auf eine Seite. Eine Seite mit nur einer Zelle gibt es, wenn ein Objekt alloziert wird, das größer als $\frac{1}{2}$ `PAGE_SIZE` und kleiner oder gleich `PAGE_SIZE` ist. Es passt nur ein Objekt dieser Größe auf eine Seite.

Die Abbildung 4.1 veranschaulicht die Begriffe Heap-Sektion, Seite, Mehrfachseite und Zelle.

## 4.2 Größenklassen und Seitenlisten

Jeder Seite oder Mehrfachseite ist ein Verwaltungsdatensatz zugeordnet, der *Seitenkopf*. Er ist nicht Teil der Seite, sondern ist in einem separaten Speicherbereich untergebracht. Der Seitenkopf enthält Informationen über die Seite, zum Beispiel aus wievielen Seiten eine Mehrfachseite besteht oder in wieviele Zellen

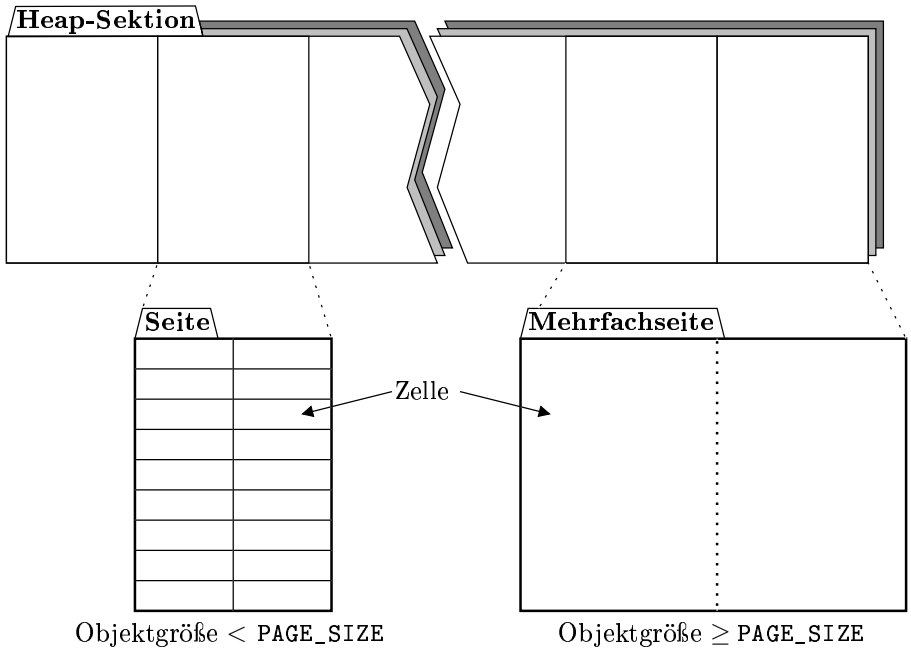


Abb. 4.1: Heap-Sektionen, Seiten und Mehrfachseiten.

die Seite eingeteilt ist. Die Seitenköpfe sind miteinander in *Seitenlisten* verknüpft.

Die Allokation erfolgt nicht nach Objekttypen, sondern nach Objektgrößen. Daher gibt es im neuen Allokator *Größenklassen*. Eine Größenklasse verwaltet alle Seiten für Objekte einer bestimmten Größe. Pro Seite gibt es nur eine Objektgröße. Die Anzahl und Abstufung der verschiedenen Größenklassen ist frei definierbar.

Jede Größenklasse wird durch eine Seitenliste repräsentiert. Die Anzahl der Größenklassen ist variabel. Es gibt zwei unterschiedliche Einteilungen: für den *benutzten Heap* und für den *unbenutzten Heap*.

Der benutzte Heap besteht aus Seiten, auf denen Objekte alloziert sind. Sie sind ganz oder teilweise belegt.

Im unbenutzten Heap sind die Seiten komplett leer. Sie waren entweder noch niemals in Benutzung oder wurden wieder freigegeben.

### 4.2.1 Benutzte Seiten

In dem Teil des Heaps, der benutzte Seiten enthält, werden die Seitenlisten nach den Objektgrößen eingeteilt, da es wichtig ist, schnell freie Zellen passender Größe zu finden. Eine Größenklasse definiert sich daher durch die Größe der Zellen einer Seite.

Die Größenklassen des benutzten Heaps sollten den Größen der Objekte angepasst werden, damit wenig Speicher verschwendet wird.

Für die Anpassung der Größenklassen stehen im benutzten Heap vier Parameter zur Verfügung, mit denen die Aufteilung der Größenklassen eingestellt werden kann:

**Kleinste Objektgröße** Die Größe des kleinsten Objekts in Bytes wird durch diesen Parameter festgelegt.

**Lineare Schrittweite** Die Schrittweite, um welche die Größenklassen bis zur unteren Schranke wachsen sollen.

**Obere Schranke** Diese Variable definiert die obere Grenze der Größenklassen. Sie wird immer auf `PAGE_SIZE` gesetzt.

Es gibt eine weitere Größenklasse im benutzten Heap, die alle Mehrfachseiten enthält. Sie ist zuständig für Objekte, die größer als `PAGE_SIZE` sind. Da es auf Mehrfachseiten keine freien Zellen gibt, ist ein schnelles durchsuchen dieser Liste nicht nötig. Eine Seitenliste reicht dafür aus.

### 4.2.2 Unbenutzte Seiten

Im unbenutzten Heap ist die Größe des zusammenhängenden Speichers ausschlaggebend. Die kleinste Einheit, die aus diesem Bereich angefordert wird, ist eine Seite. Daher werden die Größenklassen im unbenutzten Heap nach der Anzahl der zusammenhängenden Seiten eingeteilt.

Dies wird durch folgende Variablen erreicht:

**Untere Schranke** Bis zu diesem Wert gibt es für jede Anzahl eine Größenklasse.

**Lineare Schrittweite** Die Schrittweite, um die Größenklassen linear wachsen sollen.

**Obere Schranke** Dieser Parameter legt die obere Grenze der Größenklasse fest.

Ist ein zusammenhängender Speicherbereich größer als der Wert der oberen Schranke, wird er mit allen Speicherbereichen des unbenutzten Heaps, die ebenfalls diese Größe überschreiten, in einer zusätzlichen Größenklasse verwaltet.

## 4.3 Zuordnung von Heap-Zeigern zu Seitenköpfen

Nach Jones und Lins [JL96] führt ein Seitenkopf als Teil der Seite zu großen Geschwindigkeitsnachteilen beim Verwalten der Meta-Informationen. Der Grund dafür sind die oft auftretenden Miss-Penalties im Caching-Verhalten des Betriebssytems. Daher habe ich, analog zum Boehm-Kollektor [Boe04], den Seitenkopf von der Seite getrennt.

Es ist wichtig, zu jedem Heap-Objekt den zugehörigen Seitenkopf finden zu können. Der Seitenkopf wird beispielsweise für das Markieren oder die Auflösung des Objekts gebraucht. Um dies zu ermöglichen, habe ich eine Datenstruktur implementiert, die eine schnelle Zuordnung von Heap-Zeigern zu Seitenköpfen ermöglicht.

Die *Zuordnungstabelle* ist als zweistufiger Suchbaum implementiert. Die oberen 10 Bits der Speicheradresse bilden den Index in der ersten Stufe des Suchbaums. Der Eintrag des Suchbaums verweist auf die zweite Stufe, für den die

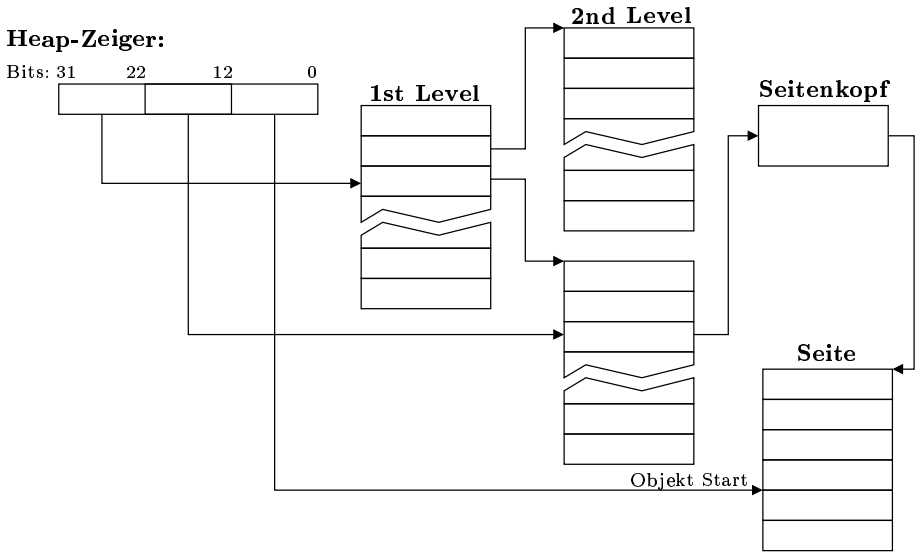


Abb. 4.2: Zuordnung von Heap-Zeigern zu Seitenköpfen.[RRSF]

nächsten 10 Bits des Heap-Zeigers den Index ergeben. Dort ist ein Verweis auf den Seitenkopf eingetragen. Die restlichen Bits des Zeigers sind der Index des Objekts innerhalb der Seite (siehe Abbildung 4.2).

Bei Rechnerarchitekturen mit mehr als 32 Bits Adressraum werden die oberen Bits mit Hilfe einer Hashfunktion in der ersten Stufe des Suchbaums nachgeschlagen.

## 4.4 Mark-Bits

Werden die Mark-Bits nicht mehr in den Heap-Objekten gespeichert, ergeben sich weitere Caching-Vorteile, da während des Markierens nicht in die Heap-Objekte geschrieben werden muss [JL96].

So kommt es nicht zu *Dirty-Caches*: der Cache-Inhalt differiert nicht vom Speicherinhalt. Das Zurückschreiben des Cache-Inhalts in den Speicher würde viel Zeit brauchen.

In jedem Seitenkopf gibt es ein Feld für die Mark-Bits. In dieses Feld passt genau ein Speicherwort. Solange es auf einer Seite weniger Zellen gibt, als Bits in ein Speicherwort passen, reicht der Speicherplatz im Seitenkopf aus. Die Mark-Bits werden im Seitenkopf gespeichert.

Sind die Objekte klein, werden auf einer Seite mehr Zellen untergebracht. Dann passen nicht mehr alle Mark-Bits in das Feld im Seitenkopf. In solchen Fällen werden die Mark-Bits in einen separaten Speicherbereich geschrieben. Das Feld im Seitenkopf hält lediglich die Referenz auf diesen Bereich.

Folgende Mark-Bit-Funktionen bilden die Schnittstelle:

```
/* mark bit functions */
void set_mark_bit (void *ptr, word value);
```

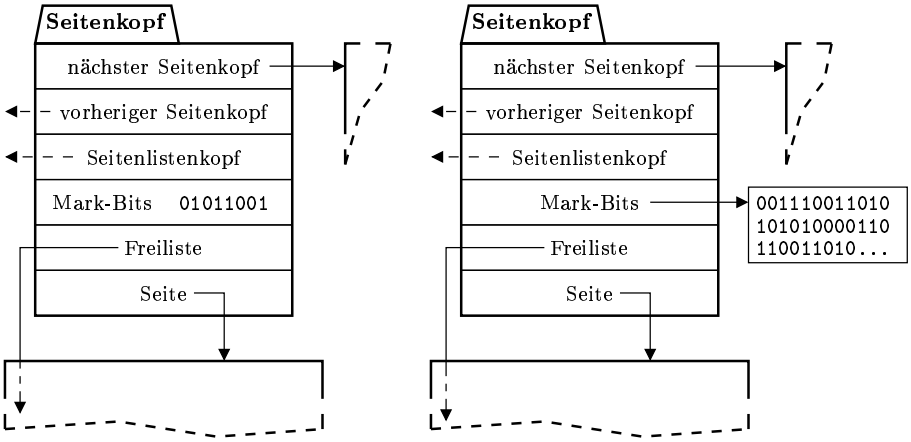


Abb. 4.3: Seitenköpfe mit Mark-Bits.

```
word get_mark_bit (void *ptr);

/* mark bit macros */
#define MARKED_P(ptr) (get_mark_bit (ptr) == 1)
#define MARK(ptr)     set_mark_bit (ptr, 1)
#define UNMARK(ptr)   set_mark_bit (ptr, 0)
```

Für eine einfachere Benutzung der Mark-Bit-Funktionen `set_mark_bit` und `get_mark_bit` stehen die Makros zur Verfügung: `MARKED_P` überprüft, ob das Objekt an Heap-Adresse `ptr` markiert ist, `MARK` markiert das Objekt und `UNMARK` entfernt die Markierung.

## 4.5 Speicher allozieren

Speicher wird durch diese Funktion angefordert:

```
void *mc_allocate (size_t size);
```

Die Funktion gibt einen Zeiger auf einen zusammenhängenden Speicherbereich der Größe `size` zurück.

Der Allokations-Algorithmus ist in Abbildung 4.4 in einem Flussdiagramm dargestellt.

Der Allokator platziert teilweise gefüllte Seiten vorne in eine Seitenliste, volle Seiten stellt der Allokator ans Ende. Das garantiert eine schnell terminierende Suche nach freien Zellen, da sich alle Seiten, auf denen Platz für neue Objekte sind, vorne befinden. Kommen volle Seiten am Anfang der Liste hintereinander, ist die komplette Größenklasse voll: Eine neue Seite muss hinzugefügt werden.

## 4.6 Heap erweitern

Um den Heap zu vergrößern, wird ein großes Stück zusammenhängender Speicher vom Betriebssystem angefordert. Im neuen Allokator heißt ein solches Stück *Heap-Sektion*.

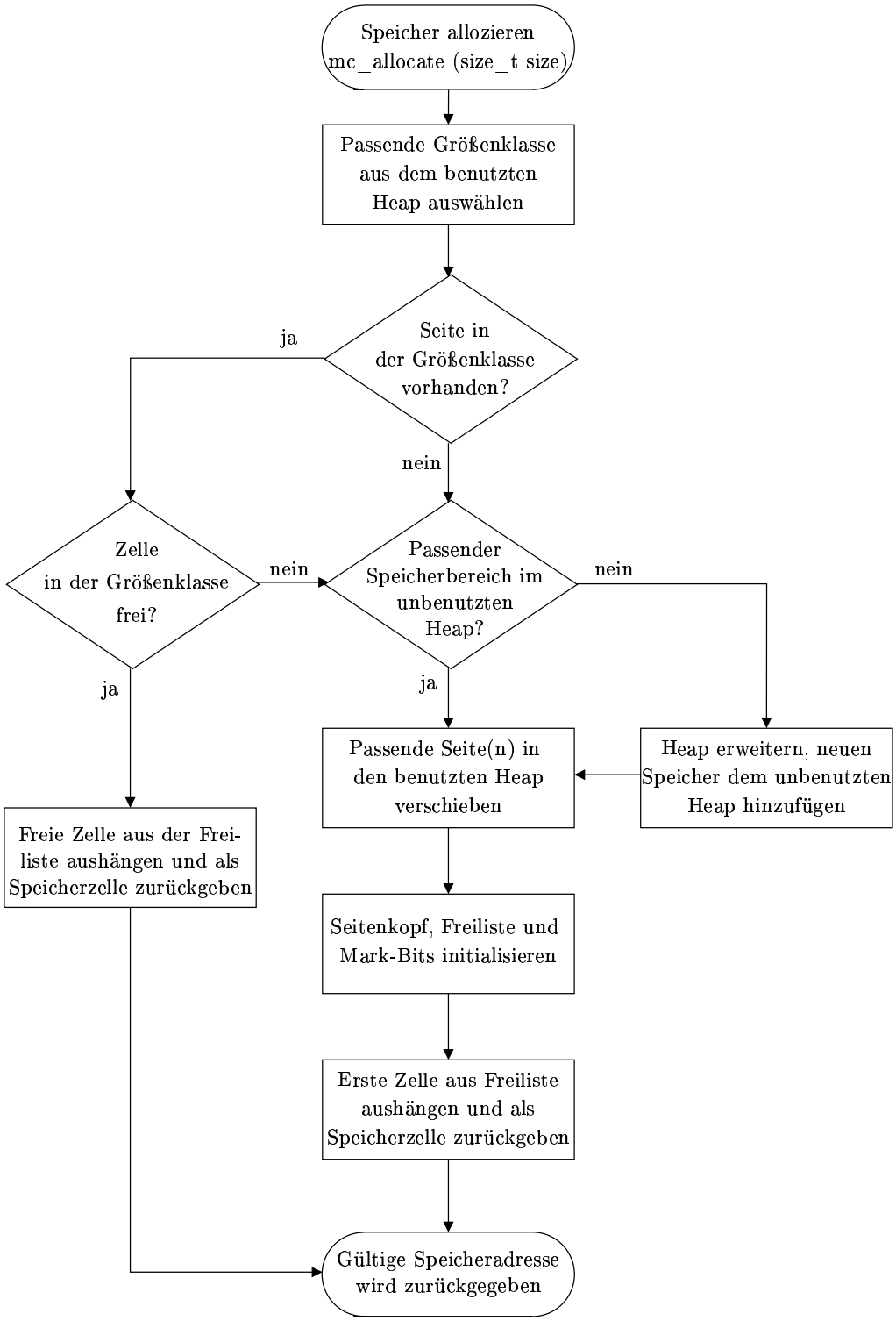


Abb. 4.4: Flussdiagramm zur Speicherallozierung.

Wie groß eine neue Heap-Sektion und somit das Wachstum des Heaps pro Erweiterung ist, kann über Parameter festgelegt werden:

**Minimales Heap-Wachstum pro Heap-Erweiterung** Dieser Parameter definiert die untere Grenze des Heap-Wachstums in Seiten. Zu Beginn hat der Heap diese Größe. Im Programm ist dafür folgende Makro-Definition anzupassen:

```
#define MIN_HEAP_INCREASE          16
```

**Maximales Heap-Wachstum pro Heap-Erweiterung** Dieser Parameter legt die maximale Anzahl der Seiten für ein Heap-Wachstum fest. Das Heap-Wachstum pro Heap-Erweiterung ist konfigurierbar durch dieses Makro:

```
#define MAX_HEAP_INCREASE          256
```

**Wachstumsdivisor** Das Wachstum pro Heap-Erweiterung wird über folgende Formel bestimmt:

$$\text{Heap-Wachstum} = \text{angeforderte Seiten} + \frac{\text{aktuelle Heap-Größe}}{\texttt{PAGE\_SIZE} * \text{Wachstumsdivisor}}$$

Die *aktuelle Heap-Größe* und `PAGE_SIZE` sind in Bytes, *Heap-Wachstum* und *angeforderte Seiten* sind in Vielfachen von Seiten angegeben.

Das Heap-Wachstum bleibt in jedem Fall zwischen unterer und oberer Grenze. Der Wachstumsdivisor kann im Programm angepasst werden:

```
#define HEAP_GROWTH_DIVISOR          3
```

Diese Vorgehensweise verringert die Anzahl von Heap-Sektionen. Je größer der Heap wächst, desto größer sind die neu angeforderten Heap-Sektionen. Werden alle Seiten einer Heap-Sektion freigegeben, wird ihr Speicherbereich an das Betriebssystem zurückgegeben.

## 4.7 Speicher freigeben

In XEmacs wird an einigen Stellen eine explizite Freigabe von lokal angelegten Lisp-Objekten vorgenommen.

Um dies zu ermöglichem, steht im neuen Allokator folgende Funktion zur Verfügung:

```
void mc_free (void *ptr);
```

Diese Funktion wird auch intern vom Allokator benutzt, um während der Sweep-Phase unmarkierte Objekte aufzulösen. In Abbildung 4.5 ist die Speicherfreigabe als Flussdiagramm dargestellt.

Wird das letzte Objekt einer Seite freigegeben, wird die nun komplett leere Seite in den unbenutzten Heap übergeben. Der Allokator versucht, diese Seite mit anderen freien Seiten zu verschmelzen, um ein möglichst großes Stück zusammenhängenden Speichers zu erhalten. Dazu wird überprüft, ob sich die benachbarten Seiten ebenfalls im unbenutzten Heap befinden.

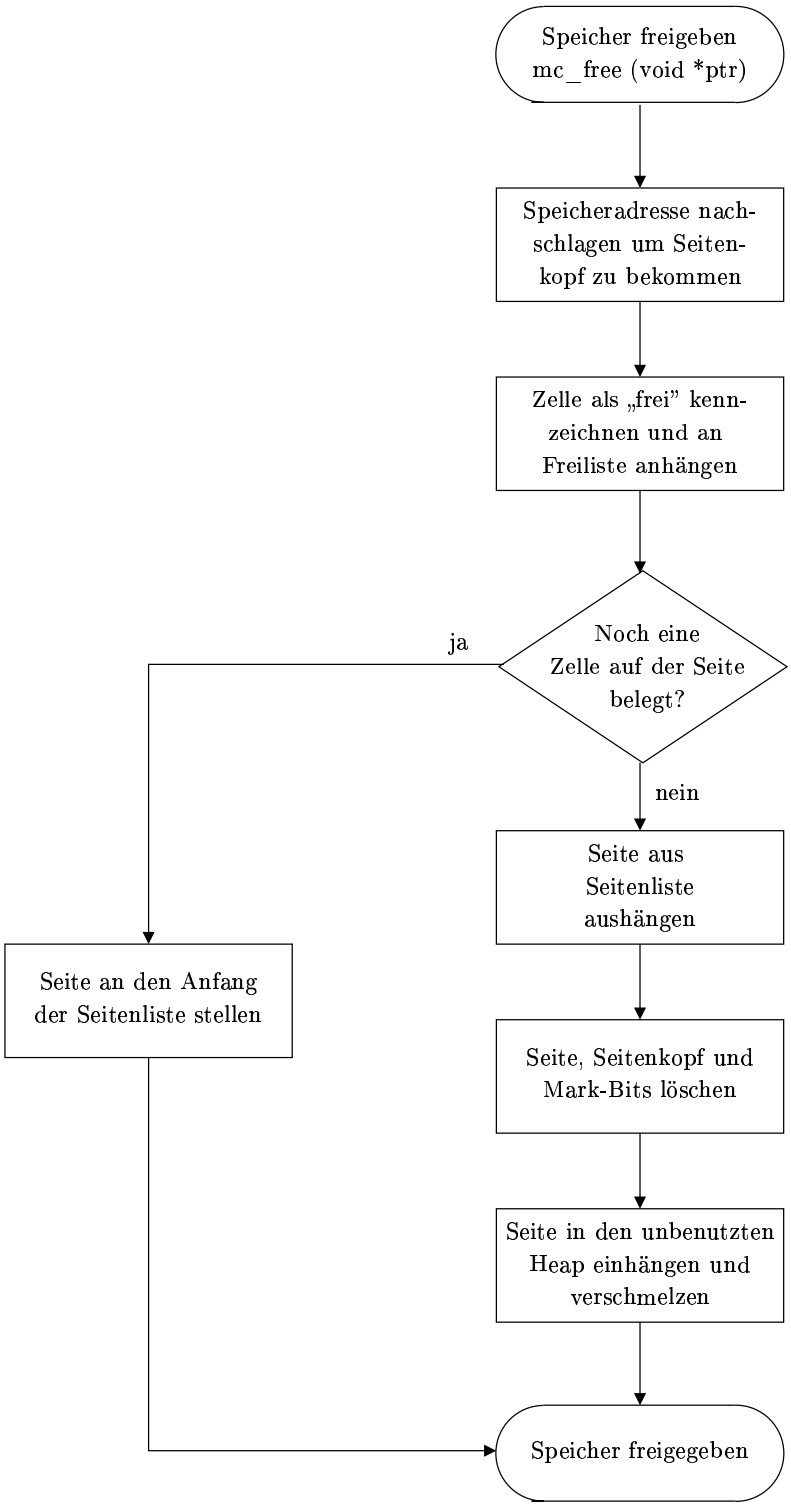


Abb. 4.5: Flussdiagramm zur Speicherfreigabe.

Der resultierende Speicherbereich wird in die entsprechende Größenklasse des unbenutzten Bereichs eingehängt und steht für spätere Allokationen zur Verfügung. Ist durch Verschmelzung eine vollständige Heap-Sektion entstanden, wird diese wieder ans Betriebssystem zurückgegeben.

## 4.8 Sweep-Phase

Der Allokator stellt eine Funktion für die Sweep-Phase bereit:

```
void mc_sweep (void);
```

Die Sweep-Funktion besucht alle Objekte im benutzten Heap und löst unmarkierte Objekte auf.

# Kapitel 5

# Änderungen an XEmacs

In diesem Kapitel beschreibe ich die Änderungen, die am XEmacs-Programmtext nötig waren, um den neuen Allokator einzubauen.

Die aktuell vorliegende Version ist noch in der Entwicklungsphase. Um die Änderungen an XEmacs so lokal wie möglich zu halten und um die Funktionsweise des bisherigen Allokators zu erhalten, habe ich die alte Schnittstelle zum Allokator vorerst übernommen. Die vorhandenen Allokator-Funktionen bilden die Einsprungspunkte zum neuen Allokator.

## 5.1 Konfiguration und Präprozessor

Durch Präprozessor-Anweisungen des C-Compilers ist der neue Allokator zu- und abschaltbar. Code, der den neuen Allokator implementiert, habe ich in `ifdef`-Anweisungen eingeschlossen.

```
#ifdef MC_ALLOC
```

*Programmcode des neuen Allokators*

```
#else /* not MC_ALLOC */
```

*Programmcode des alten Allokators*

```
#endif /* not MC_ALLOC */
```

Um den neuen Allokator anzuschalten, muss das Präprozessor-Symbol `MC_ALLOC` definiert sein. Durch Angabe des Parameters `-mc-alloc` wird XEmacs dafür konfiguriert:

```
./configure --mc-alloc
```

Der XEmacs kompiliert mit den Änderungen und den Dateien `mc-alloc.c` und `mc-alloc.h`, die den Allokator implementieren.

## 5.2 Allokation von Lrecords

Die FROB-Block-Freilisten für jeden einzelnen Lrecord-Typ konnten komplett abgeschafft werden. Vorläufig habe ich die Schnittstelle `ALLOCATE_FIXED_TYPE`

erhalten. Da FROB-Blöcke nicht länger verwaltet werden müssen, ist die neue Definition sehr einfach:

```
#define ALLOCATE_FIXED_TYPE(type, structtype, result) do {    \
  result = (structtype *) mc_allocate (sizeof (structtype));  \
} while (0)
```

Da der alte Speichermanager auch eine Auflösung der Objekte von Hand zulässt, war eine Anpassung von `FREE_FIXED_TYPE` notwendig:

```
#define FREE_FIXED_TYPE(type, structtype, ptr) do {     \
  MC_ALLOC_CALL_FINALIZER (ptr);                        \
  mc_free (ptr);                                        \
} while (0)
```

Die neue, einfachere Schnittstelle wird auch hier deutlich: der Aufruf wird an `mc_free` weitergegeben. Die Verwaltung der Freilisten erfolgt Allokator-Intern, der Klient hat damit nichts mehr zu tun.

## 5.3 Allokation von Lcrecords

Die bisherige Schnittstelle für die Allokation von Lcrecords habe ich ebenfalls erhalten. Die Allokations-Funktionen reichen die Anfragen an `mc_alloc` weiter, die Freigabe-Funktionen an `mc_free`.

## 5.4 Markierungsphase

Die XEmacs-Makros, die zur Markierung benutzt werden, habe ich abgeändert, damit die Markierungsfunktionen des neuen Allokators benutzt werden.

Am folgenden Code-Ausschnitt kann man die Änderungen und die Verwendung der `ifdef`-Anweisungen sehen:

```
#ifdef MC_ALLOC
#define MARKED_RECORD_P(obj) MARKED_P (obj)
#define MARKED_RECORD_HEADER_P(lheader) MARKED_P (lheader)
#define MARK_RECORD_HEADER(lheader)   MARK (lheader)
#define UNMARK_RECORD_HEADER(lheader) UNMARK (lheader)
#else /* not MC_ALLOC */
#define MARKED_RECORD_P(obj) (XRECORD_LHEADER (obj)->mark)
#define MARKED_RECORD_HEADER_P(lheader) ((lheader)->mark)
#define MARK_RECORD_HEADER(lheader)   ((void) ((lheader)->mark = 1))
#define UNMARK_RECORD_HEADER(lheader) ((void) ((lheader)->mark = 0))
#endif /* not MC_ALLOC */
```

Am Markierungsalgorithmus musste ich nichts mehr ändern. Er wurde im Rahmen eines Projekts entwickelt, das auch am hiesigen Lehrstuhl beheimatet ist.

## 5.5 Sweep-Phase

Sweep-Funktionen für jeden einzelnen Lrecord-Typ sind nicht mehr notwendig. Ebensowenig eine eigene Sweep-Funktion für Lcrecords. Ein Aufruf von `mc_sweep` genügt. Dadurch hat sich die Schnittstelle zum Speichermanager deutlich vereinfacht.

# Kapitel 6

# Fazit

Mit dem neuen Allokator ist XEmacs auf dem Weg zu einer modernen automatischen Speicherverwaltung. Die Arbeiten sind jedoch noch nicht abgeschlossen.

Die Schnittstelle zum Allokator wurde deutlich vereinfacht. Da der neue Allokator zunächst an- und abschaltbar sein soll, habe ich die alte Schnittstelle an manchen Stellen belassen, um die Änderungen am Programmcode so lokal wie möglich zu halten. Dadurch wird das Funktionieren und die Stabilität des XEmacs erhalten und andere XEmacs-Entwickler müssen nicht auf eine Speicherverwaltung in Entwicklung Rücksicht nehmen.

Die bisherigen Änderungen weisen eindeutig die Richtung:

- Größenklassen ersetzen Typen-Freilisten.
- Die Allokation von Lrecords und Lcrecords ist vereinheitlicht.
- Die Mark-Bits sind von den Objekten getrennt.
- Die Sweep-Phase ist aus dem Klienten herausgelöst.
- Die Allokations-Funktionalität ist vollständig vom Klienten getrennt.

Im Moment mangelt es noch an Stabilität zur Laufzeit; XEmacs stürzt oft ab. Der teilweise unübersichtliche Programmcode, die enge Verknüpfung des Mutators mit dem bisherigen Speichermanager und die Größe und Komplexität des Programms machen eine Fehlersuche sehr schwierig und zeitaufwendig. Ich hoffe, in den nächsten Wochen die letzten Fehlerquellen zu identifizieren und zu beseitigen.

Zusätzlich muss die Funktionsweise des portablen Dumpers erweitert werden: Bisher war es nicht notwendig, Strukturen und Daten des Allokators ins Speicherabbild zu schreiben und vom Speicherabbild wiederherzustellen. Durch die komplexere Freilistenverwaltung und da die Mark-Bits außerhalb der Objekte gespeichert werden, wird dies in Zukunft nötig sein.

Der nächste Schritt ist eine Veröffentlichung der Änderungen in der XEmacs-Entwicklergemeinde. Durch die Mithilfe der Entwicklergemeinde kann der neue Allokator auf verschiedenen Systemen und mit unterschiedlichen Konfigurationen getestet und weiterentwickelt werden.

## 6.1 Zukünftige Vorhaben

Einige neue Projekte ergeben sich aus den bisherigen Änderungen:

**Entfernung der zweigeteilten Allokation von *String* und *Buffer***
Die ausgelagerten Datenteile können in die vorhandenen Lisp-Objekte integriert werden.

**Endgültige Verschmelzung von Lrecords und Lcrecords**
Eine Unterscheidung ist nicht mehr notwendig. Die Entfernung der Zweiteilung kommt dem Programmierer zu Gute: Die Erzeugung neuer Lisp-Objekte wird übersichtlicher. Zudem kann dadurch der Programmcode besser auf die Dateien verteilt werden, eine Bindung der Lrecords an den Speichermanager in `alloc.c` ist nicht mehr nötig.

**Entfernung der Schnittstelle des alten Allokators**
Die neue Schnittstelle kann ohne weitere Indirektion benutzt werden.

**Optimierung des Allokators**
Der Allokator bietet viele Optimierungsmöglichkeiten: Größenklassen für den benutzten und unbenutzten Heap können angepasst, die `PAGE_SIZE` kann verändert und das Heap-Wachstum beeinflusst werden.

**Neuer Speichermanager**
Die Implementierung eines inkrementellen Speichermanagers, der Objekte nach Generationen getrennt behandelt.

# Anhang A

# Glossar

`PAGE_SIZE` einstellbare Größe einer Seite

**Heap-Sektion** Große Stücke zusammenhängenden Speichers, die der Allokator vom Betriebssystem anfordert.

**Seite** Ein `PAGE_SIZE` großes Stück Speicher, für Objekte, die kleiner als `PAGE_SIZE` sind. Wird auf einer Heap-Sektion angelegt.

**Mehrfachseite** Besteht aus mehreren zusammenhängenden `PAGE_SIZE` großen Stücken Speicher, für Objekte, die größer als `PAGE_SIZE` sind. Wird auf einer Heap-Sektion angelegt.

**Zelle** Speicherort für ein Objekt, je nach Größe des Objekts befindet sich eine oder mehrere Zellen auf einer Seite.

**Objekt** Datenstruktur, der vom Allokator Speicher zugewiesen wird. Jedes Objekt wird in einer Zelle einer Seite gespeichert.

**Größenklasse** Eine Größenklasse enthält alle Objekte einer bestimmten Größe. Jeder Größenklasse ist eine Seitenliste zugeordnet.

**Seitenliste** Verkettet Seiten (über ihre Seitenköpfe), die für eine bestimmte Zellengröße zuständig sind, in einer Liste.

**Seitenkopf** Ist der Kopf einer Seitenliste, enthält unter anderem Informationen über die Zellengröße und die Anzahl der Seiten.

**Heap** Speicherbereich, auf dem Objekte alloziert werden. Im neuen Allokator besteht der Heap aus zwei Bereichen: benutzter und unbenutzter Heap.

**benutzter Heap** Besteht aus Seiten, auf denen Objekte alloziert sind. Sie sind ganz oder teilweise belegt.

**unbenutzter Heap** Enthält Seiten, die komplett leer sind. Sie waren entweder noch niemals in Benutzung oder wurden wieder freigegeben.

# Anhang B

# C-Interface des neuen Allokators

Folgende Funktionen sind vorhanden:

`void init_mc_allocator (void)`
: initialisiert den Allokator und erzeugt alle notwendigen Datenstrukturen

`void *mc_allocate (size_t size)`
: alloziert *size* Bytes in der entsprechenden Größenklasse und liefert einen Zeiger auf diesen Speicherbereich zürück

`void mc_free (void *ptr)`
: löst das Objekt an der Speicheradresse *ptr* auf

`void *mc_realloc (void *ptr, size_t size)`
: verändert die Größe des Speicherbereichs, auf den *ptr* zeigt. Die Adresse des neuen Speicherbereichs mit der Größe *size* wird zurückgegeben.

`void mc_finalize (void)`
: Auf Objekte, die nicht markiert sind, wird der Finalisierer des jeweiligen Objekts angewandt. Das Makro `MC_ALLOC_CALL_FINALIZER` muss definiert sein, es ruft den Finalisierer des Objekts auf.

`void mc_sweep (void)`
: Alle nicht markierten Objekte auf dem Heap werden aufgelöst.

`void set_mark_bit (void *ptr, word value)`
: setzt das Mark-Bit des Objekts an Speicheradresse *ptr* auf den Wert *value*

`word get_mark_bit (void *ptr)`
: gibt den Mark-Bit-Wert des Objekts an Speicheradresse *ptr* zurück

Für die einfache Benutzung der Mark-Bit-Funktionen werden noch folgende Makros zur Verfügung gestellt:

`MARKEDP (ptr)` Prädikat: Ist das Objekt an Speicheradresse *ptr* markiert?

`MARK (ptr)` markiert das Objekt an Speicheradresse *ptr*

`UNMARK (ptr)` löscht die Markierung des Objekts an Speicheradresse *ptr*

In XEmacs gibt es eine Finalisierungsfunktionen für den Dumper:

`void mc_finalize_for_disksave (void)` Unnötige Daten der Objekte werden vor dem Dumpen aufgelöst. Das Makro `MC_ALLOC_CALL_FINALIZER_FOR_DISKSAVE` muss definiert sein, es ruft den Disksave-Finalisierer des Objekts auf.

# Anhang C

# Datenstrukturen des neuen Allokators

- `struct mc_allocator_globals`
  enthält alle global zugänglichen Variablen.

  `word heap_size`
  : aktuelle Größe des Heaps in Byte

  `heap_sect heap_sections[MAX_HEAP_SECTS]`
  : Liste aller einzeln allozierter Heap-Sektionen. Steht nicht mehr genug Speicher zur Verfügung, wird der Heap erweitert. Dazu wird ein neues Stück zusammenhängender Speicher vom Betriebssystem angefordert. Die Größe des angeforderten Speichers hängt von der Größe des bisherigen Heaps und der Größe des benötigten Speichers ab.

  `int n_heap_sects`
  : Zahl der einzeln allozierten Heap-Sektionen

  `page_list_header *used_heap_pages[N_USED_PAGE_LISTS]`
  : Zeigt auf alle Seitenlisten, deren Seiten in Benutzung sind. Die Größen der Objekte werden in Größenklassen zusammengefaßt. Jede Größenklasse wird in einer eigenen Liste verwaltet (dies garantiert schnelle Allokation auf nur teilweise gefüllten Seiten). Objekte, die größer sind als `PAGE_SIZE` , werden in einer eigenen Liste verwaltet.

  `page_list_header *free_heap_pages[N_FREE_PAGE_LISTS]`
  : Array aller Seitenlisten, deren Seiten nicht in Benutzung sind. Zusammenhängende Seiten werden zu einer Mehrfachseite verschmolzen und in die entsprechende Liste eingehängt. In einer Liste sind jeweils gleich große Mehrfachseiten. Die Größe einer Mehrfachseite bestimmt die Freiliste, in der sie eingehängt wird.

  `level_2_lookup_tree *ptr_lookup_table[LEVEL1_SIZE]`
  : Tabelle für die Zuordnung von Heap-Zeigern zu Page-Headern

  `free_link *page_header_free_list`
  : Freiliste für zur Zeit unbenutzte Seitenköpfe (Page-Header)

`word malloced_bytes`
: zur Statistik: insgesamt allozierter Speicher des Allokators (mit allen beteiligten Datenstrukturen)

- `struct heap_sect`
  Repräsentation der Heap-Sektionen.

  `void *real_start`
  : Start der Heap-Sektion, nicht am Vielfachen der `PAGE_SIZE` ausgerichtet

  `size_t real_size`
  : Größe der Heap-Sektion, kein Vielfaches von `PAGE_SIZE`

  `void *start`
  : Start der Heap-Sektion, am Vielfachen der `PAGE_SIZE` ausgerichtet

  `size_t n_pages`
  : Größe der Heap-Sektion, Vielfaches von `PAGE_SIZE`

- `struct page_list_header`
  Listenkopf für eine Seitenliste

  `word list_type`
  : Enthält den Typ der Liste (`USED_LIST` oder `FREE_LIST`)

  `size_t size`
  : Ist diese Liste eine `USED_LIST`, enthält die Variable die Größe einer Objekt-Zelle in dieser Liste. Ist diese Liste eine `FREE_LIST`, enthält die Variable die Größe des Heap-Sektors.

  `page_header *first`
  : Referenz zum ersten Eintrag der Seitenliste

  `page_header *last`
  : Referenz zum letzten Eintrag der Seitenliste

  `free_link *markbit_free_list`
  : Werden die Mark-Bits in einem separaten Bereich gespeichert (siehe `mark_bits` in `page_header`), dann enthält `markbit_free_list` eine Freiliste der Mark-Bit-Speicherbereiche (für speziell diese Objektgröße).

- `struct page_header`
  Repräsentation des Seitenkopfs

  `page_header *next`
  : nächster Seitenkopf der Seitenliste

`page_header *next`
vorheriger Seitenkopf der Seitenliste

`page_list_header *plh`
Zeiger zum Listenkopf dieser Seitenliste
$\Rightarrow$ doppelt verlinkte Liste mit Referenz auf den Listenkopf

`free_link *free_list`
verlinkt die freien Zellen der Seite

`word n_pages`
Die Anzahl der `PAGE_SIZE` großen Seiten, die zusammen die Mehrfachseite bilden (für Objekte, die Größer als `PAGE_SIZE` sind).

`word cell_size`
Größe einer Zelle (in Bytes)

`word cells_on_page`
Gesamtzahl der Zellen auf der Seite

`word cells_used`
Zähler für die Anzahl der belegten Zellen auf der Seite

`char *mark_bits`
Verwaltet eine Seite weniger Zellen als es Bits in einem `word` gibt (meist sind es 32 Bits je `word` ), werden die Mark-Bits direkt in der `mark_bits` Variable im Seitenkopf gespeichert. Verwaltet die Seite mehr Objekte als die Anzahl der Bits pro `word` , enthält `mark_bits` einen Zeiger auf einen separaten Speicherbereich, in dem die Mark-Bits gespeichert werden.

`void *heap_space`
Zeiger auf den Anfang des eigentlichen Speicherbereichs für die Objekte

- `struct free_link`
  Freilisten-Repräsentation (einfach verlinkte Liste)

`struct lrecord_header lheader`
XEmacs-Spezifischer Vorspann jedes Lisp-Objekts

`struct free_link *next_free`
Zeiger auf das nächste Element der Freiliste

# Literaturverzeichnis


[Boe04] Hans-J. Boehm. *A garbage collector for C and C++*. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html`, 2004.

[JL96] Richard Jones und Rafael Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons Ltd., 1996.

[LAX] *Archived XEmacs Mailing Lists*. `http://list-archive.xemacs.org`.

[Lor03] Tom Lord. *Working on a new GC*. Nachricht an comp.lang.scheme, August 2003.

[LSM+] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram und Ulrich Drepper. *The GNU C Library Reference Manual*. `http://www.gnu.org/software/libc/manual/`.

[Rei01] Richard Reingruber. *Alternative Speicherverwaltung für XEmacs*. Diplomarbeit, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2001.

[RRSF] Gustavo Rodriguez-Rivera, Mike Spertus und Charles Filterman. *Conservative Garbage Collection for General Memory Allocators*. `http://www.cs.purdue.edu/homes/grr/ismm2000.pdf`.

[SW] Richard Stallman und Ben Wing. *XEmacs User's Manual*. In der XEmacs-Distribution enthalten.

[WBN+] Ben Wing, Martin Buchholz, Hrvoje Niksic, Matthias Neubauer und Olivier Galibiert. *XEmacs Internals Manual*. In der XEmacs-Distribution enthalten.

[WLLS] Ben Wing, Bil Lewis, Dan LaLiberte und Richard Stallman. *XEmacs Lisp Reference Manual*. In der XEmacs-Distribution enthalten.