Reference Counting (1)

- Prinzip: jeder Chunk *C* bekommt einen Zähler
  - Initialisierung des Zählers mit 1 bei der ersten Allokation
  - Inkrementierung, wenn eine Referenz auf *C* kopiert wird
  - Dekrementierung, wenn eine Referenz auf *C* eliminiert wird
  - wenn der Zähler auf Null fällt, kann *C* entsorgt werden

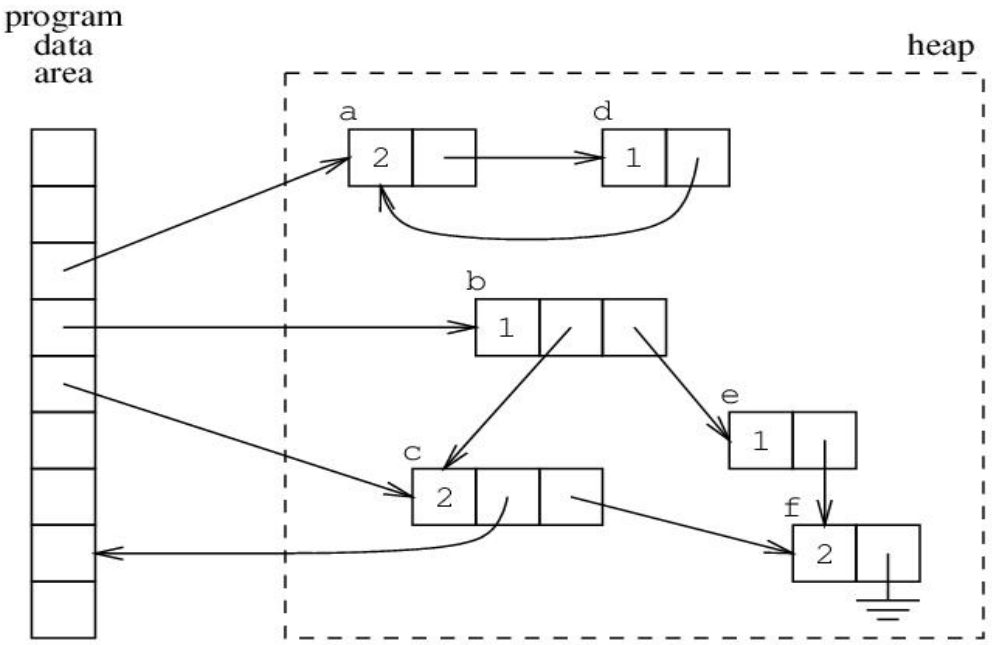

Figure 5.7 Chunks with reference count in a heap.

Reference Counting (2)

- Referenzen: Duplikation und Elimination
  - Kopie einer Referenzvariablen führt zur Erhöhung der Anzahl der Referenzen
  - nicht nur Assignments führen zu Kopien
  - nicht alle Referenzen weisen in den Heap
  - Verschwinden einer Referenz im Quellprogramm i.d.R. implizit
    - eine Zuweisung überschreibt die Referenz
    - der die Referenz enthaltene Block wird verlassen

Reference Counting (3)

- Bsp.: Referenz auf `b` wird gelöscht

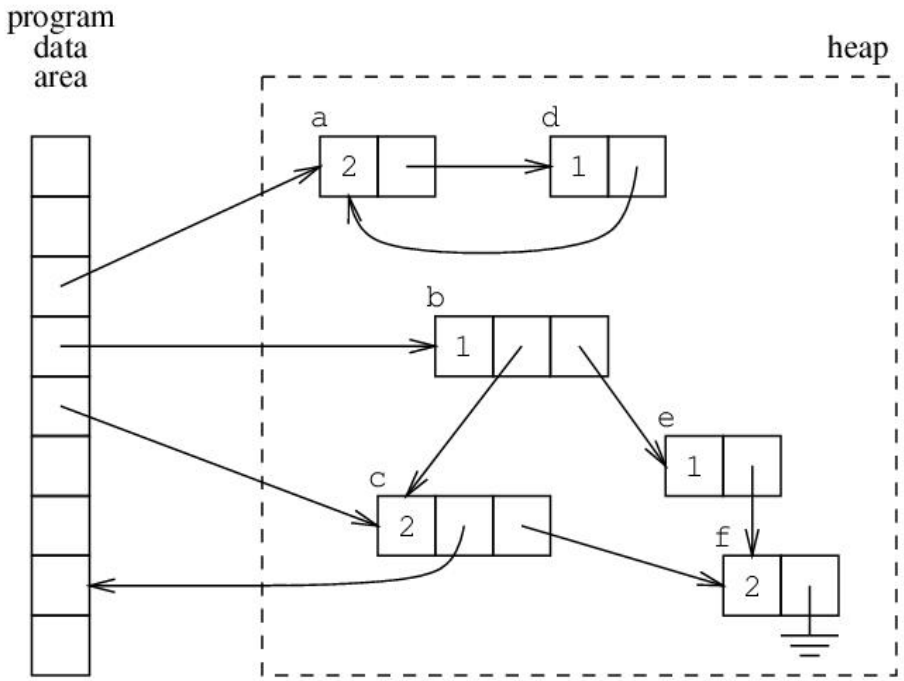

Figure 5.7 Chunks with reference count in a heap.

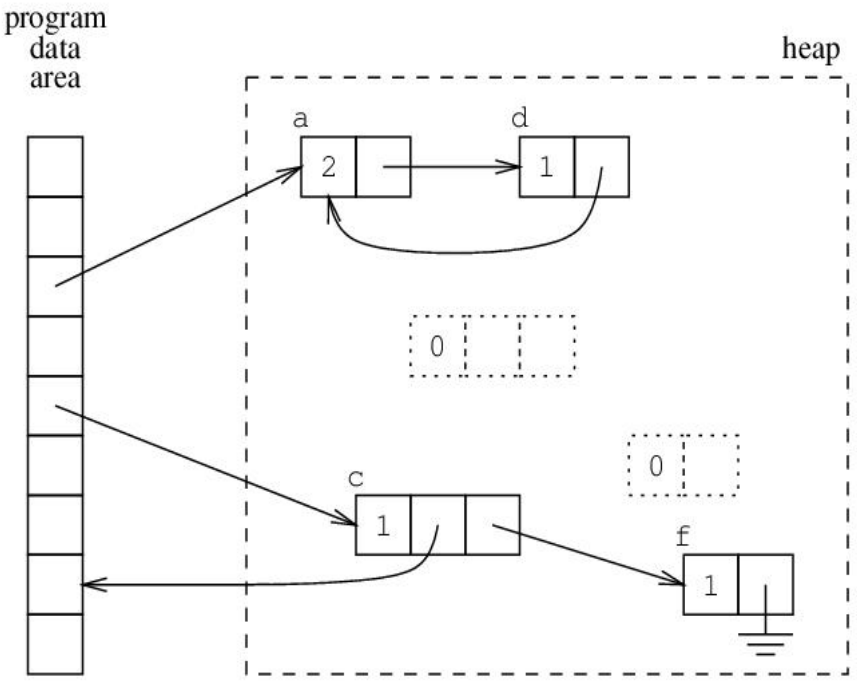

Figure 5.8 Result of removing the reference to chunk `b` in Figure 5.7.

Reference Counting (4)

- Aktionen bei Pointerzuweisung `p:=q`

```
IF Points into the heap (q):
    Increment q .reference count;
IF Points into the heap (p):
    Decrement p .reference count;
    IF p .reference count = 0:
        Free recursively depending on reference counts (p);
SET p TO q;
```

Figure 5.9 Code to be generated for the pointer assignment `p:=q`.

Reference Counting (5)

- Heapbereinigung:
  - Aufgabe: Entsorgung eines freigewordenen Chunks
    - Dekrementierung der Zähler aller Referenzen im Chunk
    - Freigabe zu Müll gewordener Chunks (Abb. 5.8)
  - Problem: Rekursion im Kollektor
    - erfordert unbeschränkten Stackraum
  - Lösungen
    - am besten: pointer reversal (später: Fol. 14/11-13)
    - kleine Verbesserung: Elimination von Endrekursion (nächste Folie)
- Nachteile:
  - versagt bei der Elimination zyklischer Strukturen
  - belastet alle Zeigeroperationen mit Laufzeitoverhead
  - fragmentiert den Heap (i.d.R. keine Kompaktifizierung)

Reference Counting (6)

```
PROCEDURE Free recursively depending on reference counts(Pointer);
    WHILE Pointer /= No chunk:
        IF NOT Points into the heap (Pointer): RETURN;
        IF NOT Pointer .reference count = 0: RETURN;

        FOR EACH Index IN 1 .. Pointer .number of pointers - 1:
            Decrement Pointer .pointer [Index] .reference count
            Free recursively depending on reference counts
                (Pointer .pointer [Index]);

        SET Aux pointer TO Pointer;
        IF Pointer .number of pointers = 0:
            SET Pointer TO No chunk;
        ELSE Pointer .number of pointers > 0:
            SET Pointer TO
                Pointer .pointer [Pointer .number of pointers];
            Decrement Pointer .reference count
        Free chunk(Aux pointer);    // the actual freeing operation
```

Figure 5.10 Recursively freeing chunks while avoiding tail recursion.

fehlt im Buch

Reference Counting (7)

- Reference Counting versagt bei zyklischen Referenzen:

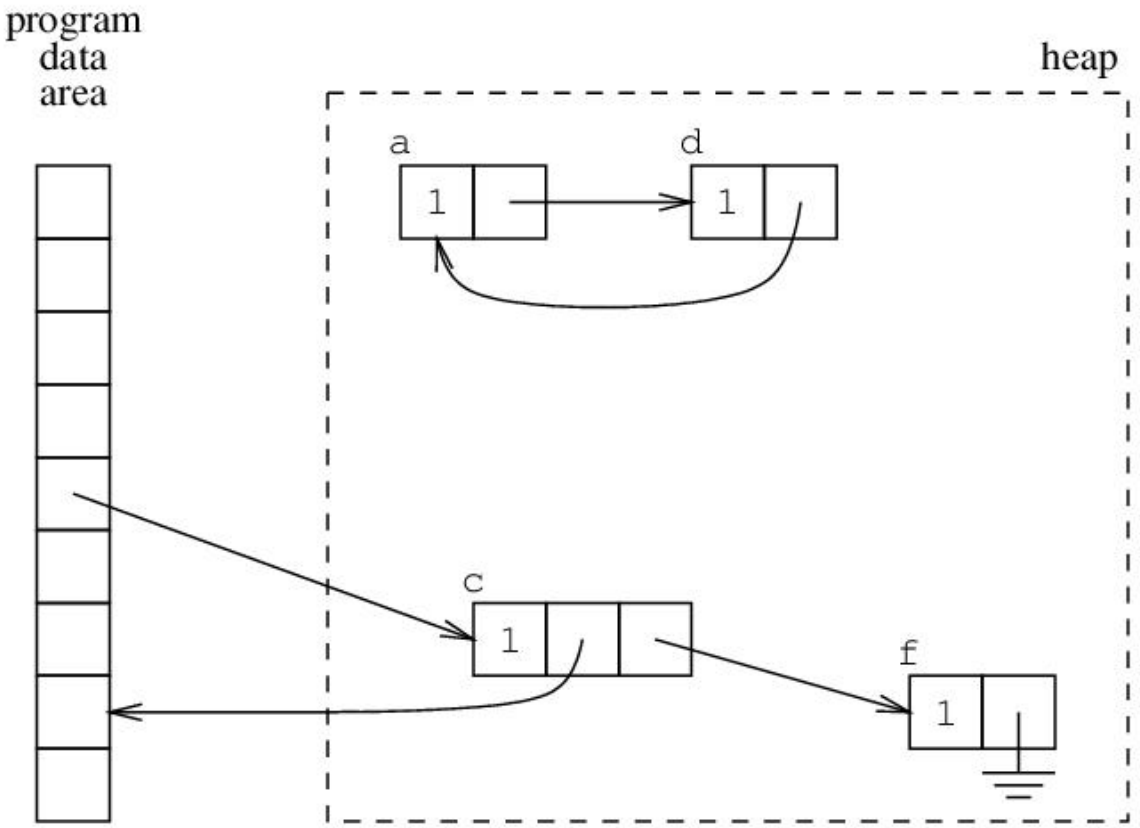

Figure 5.11 Reference counting fails to identify circular garbage.

Mark and Scan

- Mark-Phase:
  - verfolge Zeiger auf Chunks rekursiv ausgehend vom Root Set
  - Voraussetzung: Zeigerstellen vom Compiler gekennzeichnet
  - markiere alle erreichbaren, noch nicht markierten Chunks durch Tiefensuche
- Scan-Phase:
  - gehe die Chunks in der Reihenfolge im Speicher durch und gib die nicht markierten Chunks frei

Mark and Scan, Markierung (1)

- Implementierung:
  - Problem: Rekursion benötigt unbeschränkten Stack
  - Lösungsidee: verteile Stack über die markierten Chunks
  - Information im Chunkkopf (Abb. 5.12)
    - *marked bit*: wird am Anfang des Chunkbesuchs gesetzt
    - *free bit*: mit Null (false) initialisiert
    - *parent pointer*: Rückwärtszeiger auf den Vorgängerchunk
    - *pointer counter*: Zahl schon besuchter Referenzen im Chunk
  - Overhead pro Chunk: eine Referenz + ein Zähler + 1 Bit (in LISP unakzeptabel)
  - Lösung ohne Overhead: pointer reversal (Fol. 14/11-13)

Mark and Scan, Markierung (2)

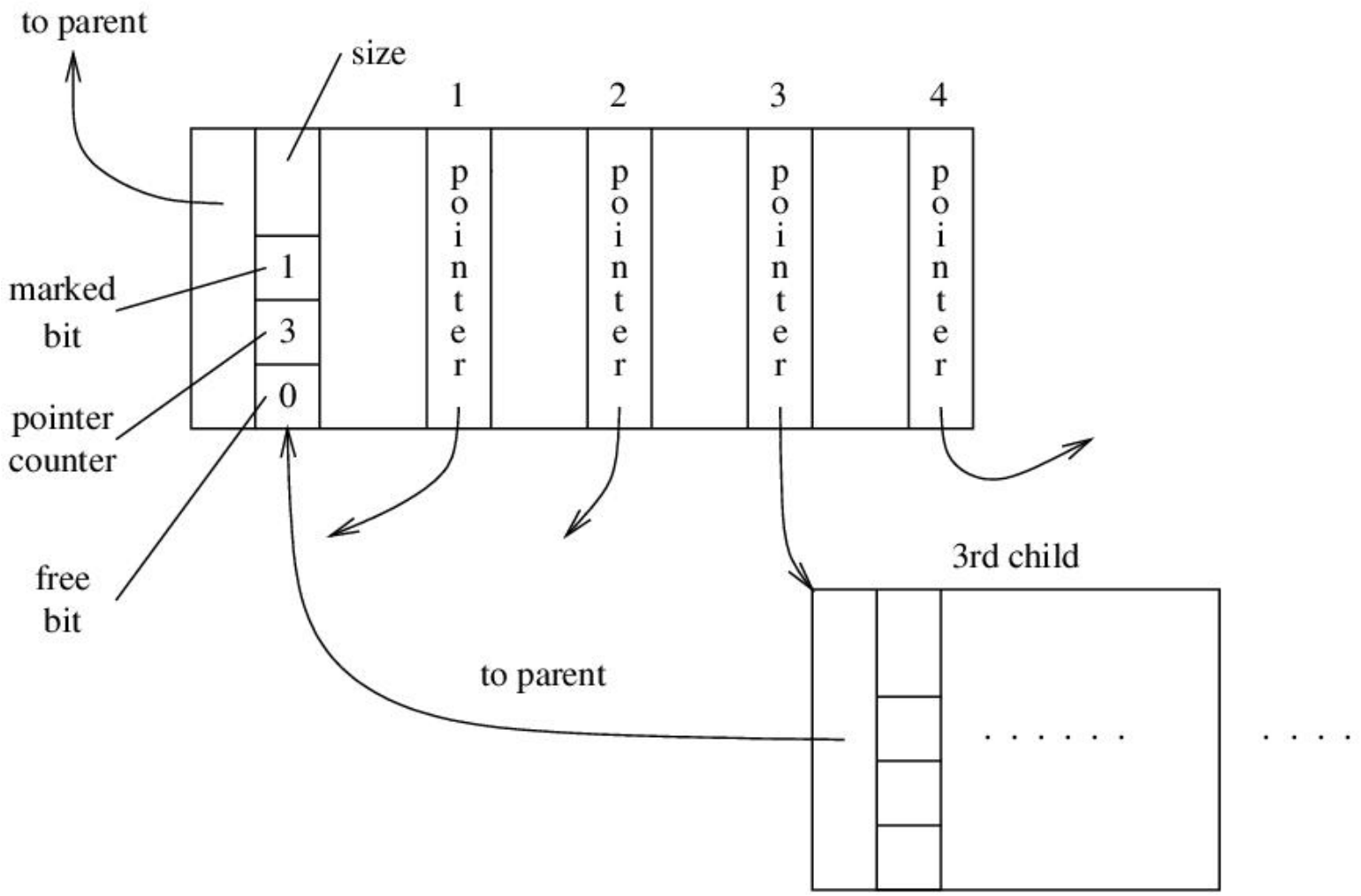

Figure 5.12 Marking the third child of a chunk.

Pointer-Reversal (1)

- Prinzip: Wiederverwendung redundanten Speichers
  - Algorithmus benutzt zwei separate Zeiger:
    - Chunk pointer: zeigt auf gerade besuchten Chunk
    - Parent pointer: zeigt auf dessen Vorgängerchunk
  - Szenario:
    - Zeigerkette: Chunk P → Chunk C → Chunk D
    - Parent pointer → P
    - Chunk pointer → C
  - nächster Schritt (Abschreiten der Referenz auf D in C):
    - Parent pointer → C
    - Chunk pointer → D
  - Beobachtung: zwei Referenzen auf D (Chunk pointer und in C)
  - Konsequenz: verwende den Slot für die Referenz auf D in C für den Zeiger auf den Vorgänger (statt Feld im Chunkkopf)

Pointer-Reversal (2)

- Implementierung:
  - rotiere Information über drei Zeiger:
    - Parent pointer, Chunk pointer, Slot für Zeiger auf D in C
  - vorwärts beim Abstieg:

```
// Chunk_pointer → C
      Old_parent_pointer := Parent_pointer
      Parent_pointer := Chunk_pointer
// Parent_pointer → C
      Chunk_pointer := n-th pointer field in C
      n-th pointer field in C := Old_parent_pointer
```

  - rückwärts beim Aufstieg:

```
// Parent_pointer → C
      Old_parent_pointer := Parent_pointer
// Old_parent_pointer → C
      Parent_pointer := n-th pointer field in D
      n-th pointer field in C := Chunk_pointer
      Chunk_pointer := Old_parent_pointer
// Chunk_pointer → C
```

Pointer-Reversal (3)

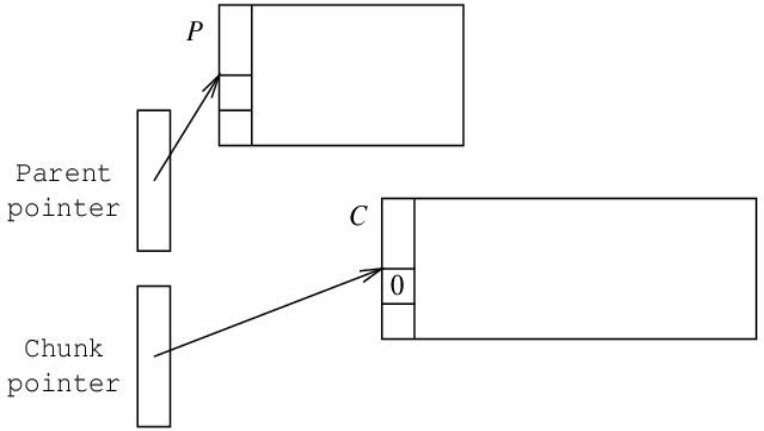

Figure 5.13 The Schorr and Waite algorithm, arriving at C.

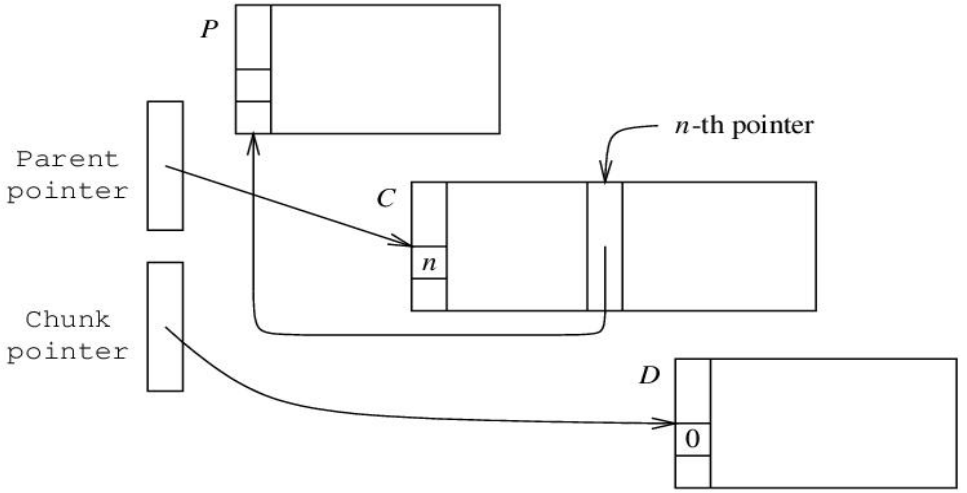

Figure 5.14 Moving to D.

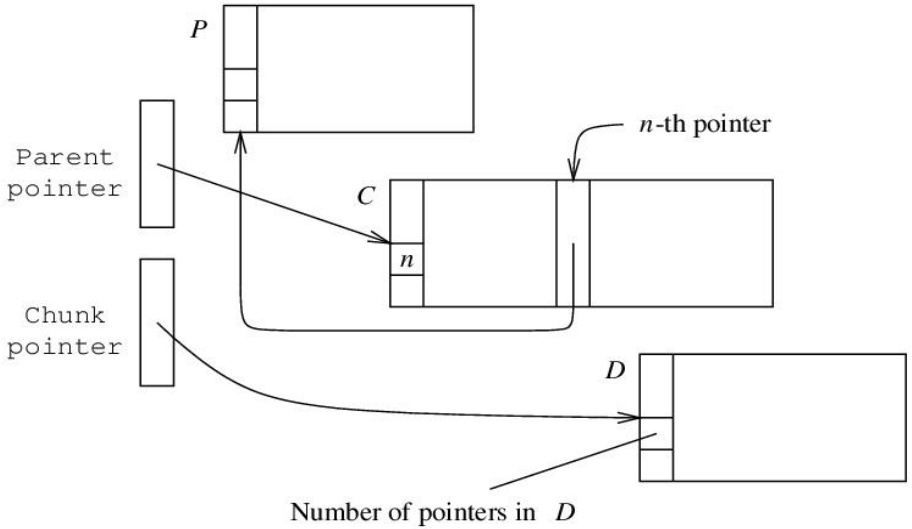

Figure 5.15 About to return from D.

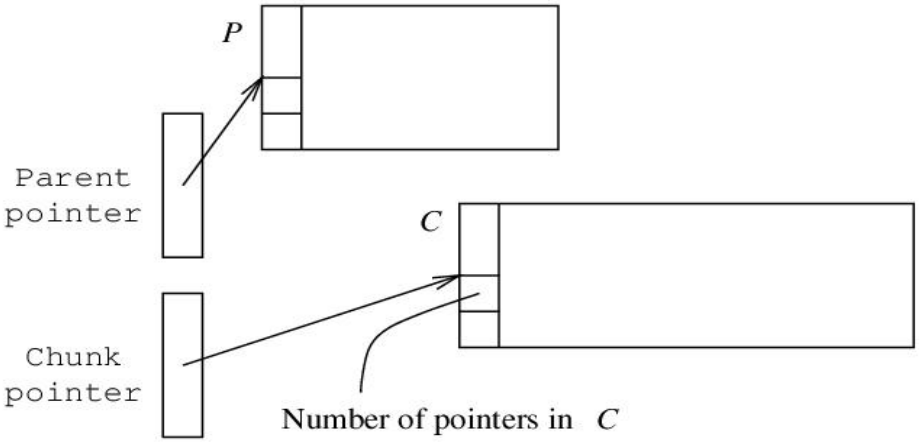

Figure 5.16 About to return from C.

Mark and Scan, Freigabe

- Vorgehen
  - durchlaufe alle Chunks im Heap
    (Beginn des nächsten Chunks über Chunkgröße bestimmbar)
  - lösche das Markierungbit markierter Chunks
  - vereinige benachbarte freie Chunks
    (mittels entsprechendem Update im Kopf des ersten Chunks)
- Ergebnis
  - Folgen belegter Chunks getrennt durch nur einen freien Chunk
- Nachteil
  - alle Chunks müssen besucht werden, nicht nur die erreichbaren
    (bei Garbage Collection ist i.d.R. der Hauptteil des Heaps Müll)
- Kompaktifizierung des freien Heapbereichs
  - durch Verschieben besetzter Chunks (Fol. 14/24-27)

Concurrent Garbage Collection (1)

- Algorithmus: [Dijkstra et al.: CACM 21(11), Nov. 1978, 966-975]
- Beweis: [David Gries: CACM 20(12), Dez. 1977, 921-930]
- Idee: zwei parallele Prozesse
  - Mutator: Anwenderprogramm
  - Kollektor: endloser Mark-and-Scan Algorithmus
- Einschränkungen
  - Betrachtung von LISP-Datenstrukturen
  - homogene Chunks (CAR-Zeiger, CDR-Zeiger)
  - zwei-elementiges Root Set:
    - ein Zeiger auf die Anwenderdatenstruktur
    - ein Zeiger auf die Freiliste
- Chunkfarben (dritte Farbe zur Mutator/Kollektor-Kooperation)
  - schwarz (erreichbar, wird nur vom Kollektor festgelegt)
  - grau (erreichbar, aber noch nicht vom Kollektor bearbeitet)
  - weiss (nicht erreichbar, wenn kein grauer Chunk existiert)

Concurrent Garbage Collection (2)

- Mutator
  - wartet, bis die Freiliste mindestens zwei Chunks enthält (vermeidet Update-Konflikt mit Kollektor)
  - entnimmt Chunk vom Kopf der Freiliste
  - schattiert ihn (weiß→grau, grau→grau, schwarz→schwarz)
- Kollektor: endlose Mark-and-Scan Zyklus
  - Mark-Phase
    - zu Beginn existieren keine schwarzen Chunks
    - schattiere Root Set (danach sind seine Chunks grau)
    - für jeden grauen Chunk
      - schattiere die direkten Nachfolger
      - schwärze den Chunk selbst
    - erreichbarer Heapteil wird schwarz mit grauer Front
    - am Ende: alles Erreichbare schwarz, alles andere weiß
  - Scan-Phase
    - hänge die weißen Chunks ans Ende der Freiliste
    - mache alle schwarzen Chunks weiß

Concurrent Garbage Collection (3)

- Interaktion von Mutator und Kollektor
  1. Mutator nimmt einen Chunk von der Freiliste und schattiert ihn der Chunk ist grau oder schwarz
  2. Mutator entfernt seine Referenz von dem Chunk
  3. Kollektor schwärzt Chunk
  4. Anfang nächster Zyklus: Kollektor initialisiert Chunk mit weiß
  5. Chunk bleibt weiß (nicht erreichbar)
  6. Chunk wird an die Freiliste angehängt
- Beweis
  - Invariante: kein direkter Zeiger von schwarz nach weiß
  - jeder Chunk wird spätestens beim zweiten Kollektorzyklus, nachdem er unerreichbar geworden ist, eingesammelt
- Kooperation von Mutator und Kollektor
  - ohne gegenseitigen Ausschluss
  - nur mit Test&Set-Operation

Two-Space Copying (1)

- Idee:
  - teile den Heap in zwei Teile
    - From-Space: nimmt Blöcke auf
    - To-Space: bleibt zunächst leer
  - wenn kein Platz mehr im From-Space ist
    - übertrage die erreichbaren Chunks kompakt in den To-Space

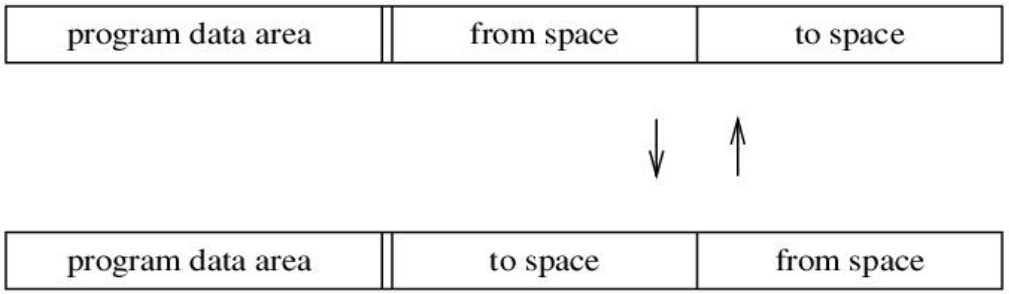

Figure 5.17 Memory layout for two-space copying.

Two-Space Copying (2)

- Implementierung:
  1. übertrage die vom Root-Set erreichbaren Chunks bündig (Kompaktifizierung)
     - lasse in den Kopien auftretende Zeiger noch unberührt
     - gib den Originalen eine Marke und Verweis auf die Kopie
  2. scanne die Folge der Chunks im To-Space, für jeden Zeiger im aktuellen Chunk
     - referenziertes Objekt schon kopiert: biege Zeiger um auf die Kopie im To-Space
     - sonst: übertrage referenziertes Objekt in To-Space (s.o.)
  3. vertausche die Rollen von From- und To-Space

Two-Space Copying (3)

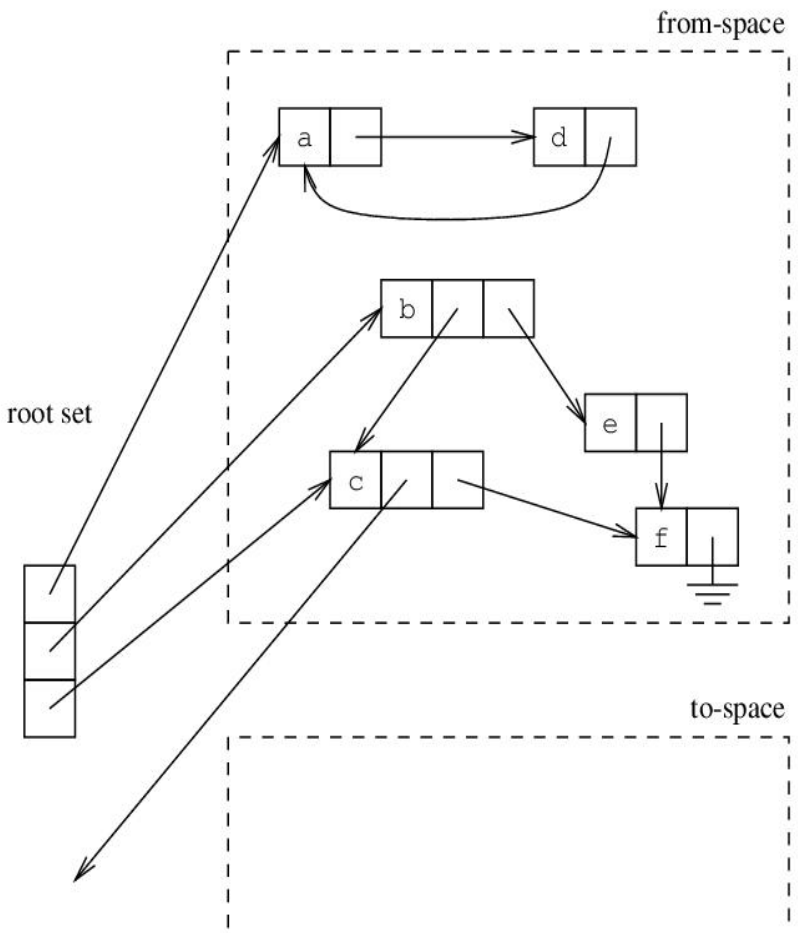

Figure 5.18 Initial situation in two-space copying.

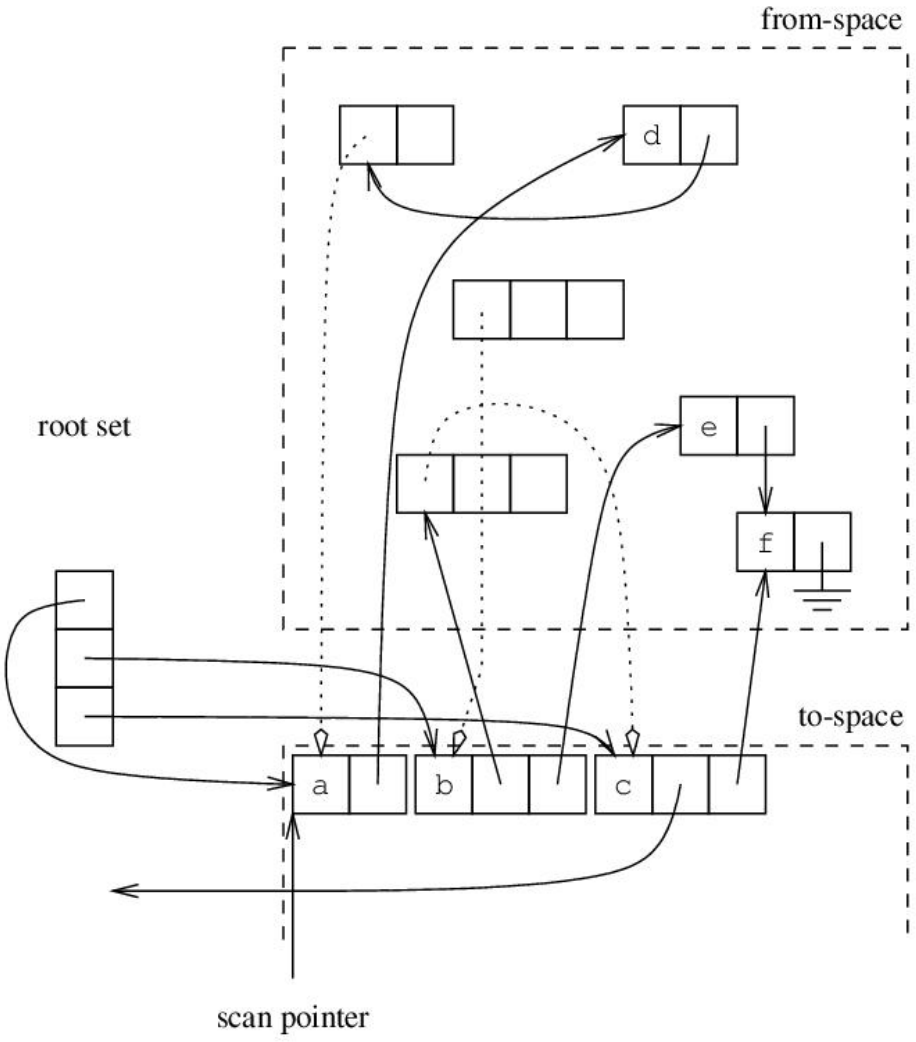

Figure 5.19 Snapshot after copying the first level.

Two-Space Copying (4)

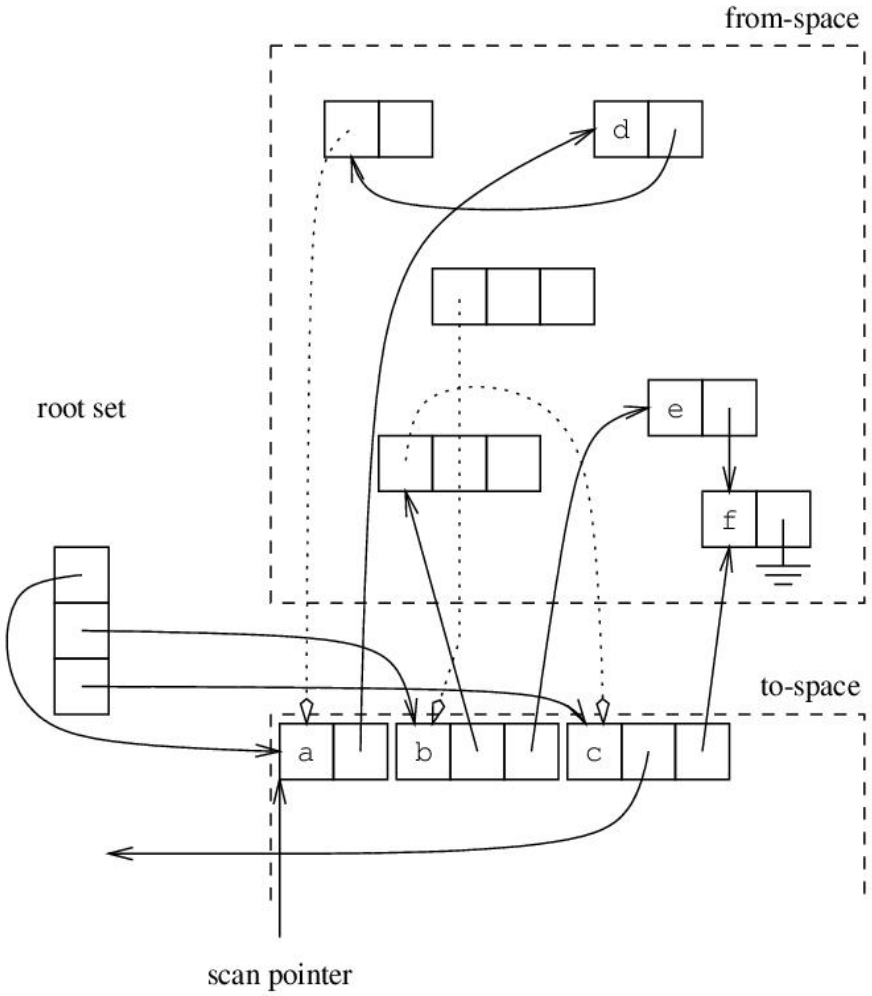

Figure 5.19 Snapshot after copying the first level.

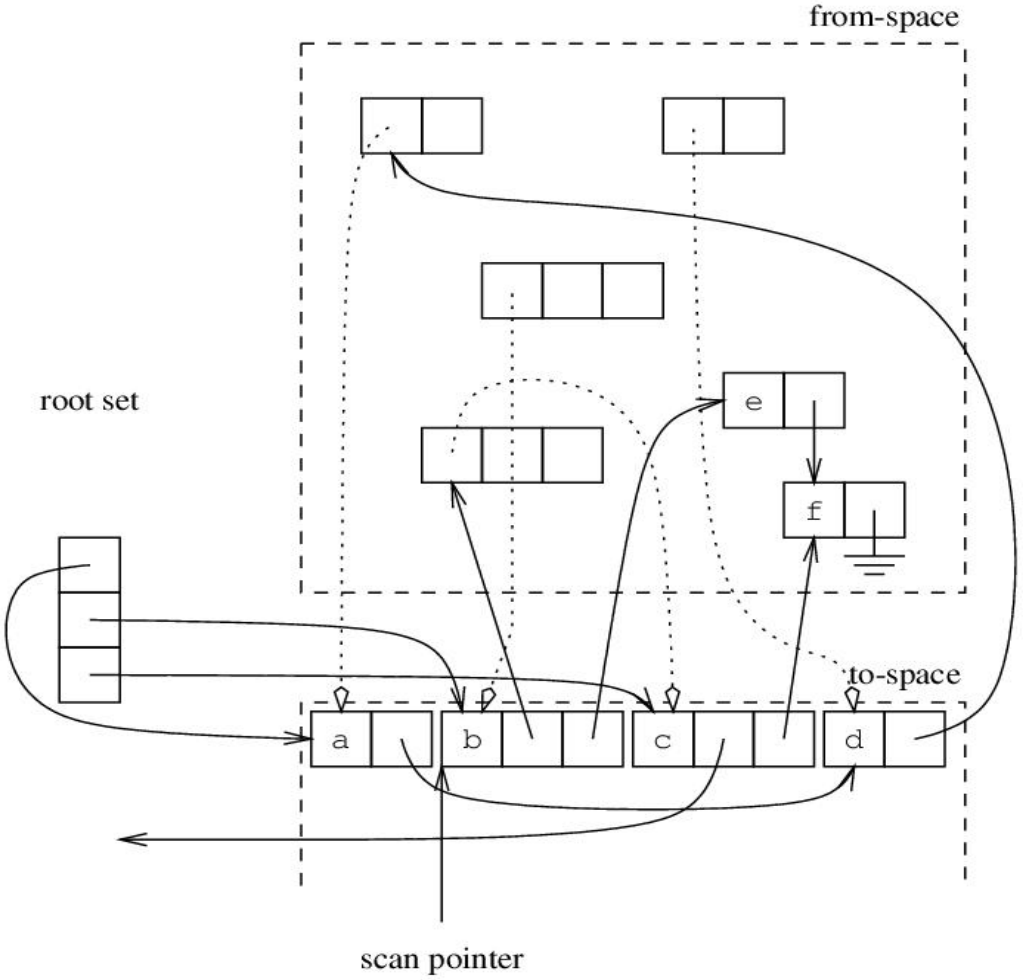

Figure 5.20 Snapshot after having scanned one chunk.

Two-Space Copying (5)

- Vorteile
  - benötigt keinen Stack
  - wenig Laufzeitoverhead
  - implizite Kompaktifizierung
- Nachteile
  - braucht doppelten Speicherplatz
  - schlechte Leistung, wenn Heap über längere Zeit fast voll

Kompaktifizierung (1)

- Idee:
  - schiebe alle besetzten Chunks zusammen
  - fasse den freien Teil des Heaps zu einem Chunk zusammen
  - am effektivsten nach einer Kollektion
- Implementierung: drei Durchläufe des Heaps
  1. Adressberechnung (Abb. 5.21)
     - berechne für jeden besetzten Chunk die neue Position aus Position und Größe des Vorgängers
  2. Zeiger-Update (Abb. 5.22)
     - aktualisiere in den Chunks alle Zeiger auf andere Chunks
  3. Verschiebung der Chunks (Abb. 5.23)
     - problemlos, wenn von links nach rechts im Speicher
     - der letzte Chunk ist der einzig freie

Kompaktifizierung (2)

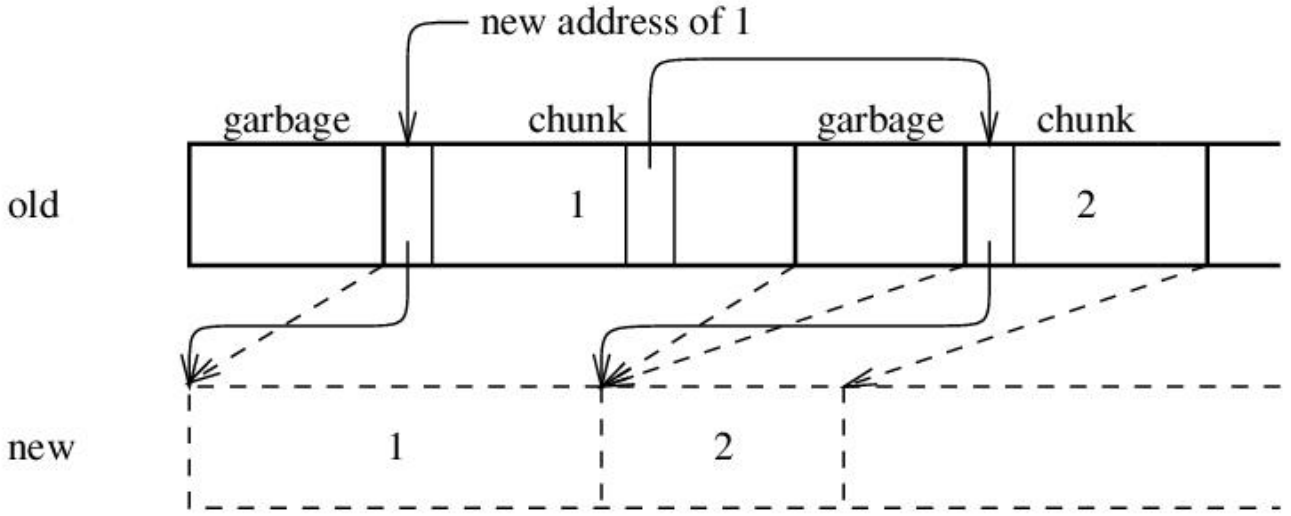

Figure 5.21 Address calculation during compaction.

Kompaktifizierung (3)

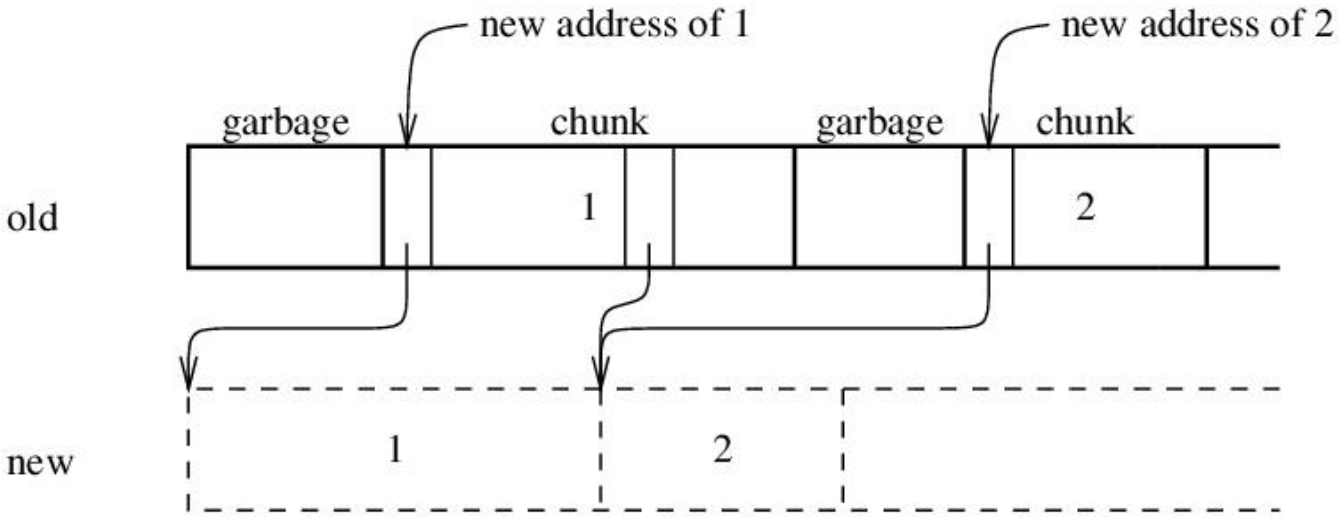

Figure 5.22 Pointer update during compaction.

Kompaktifizierung (4)

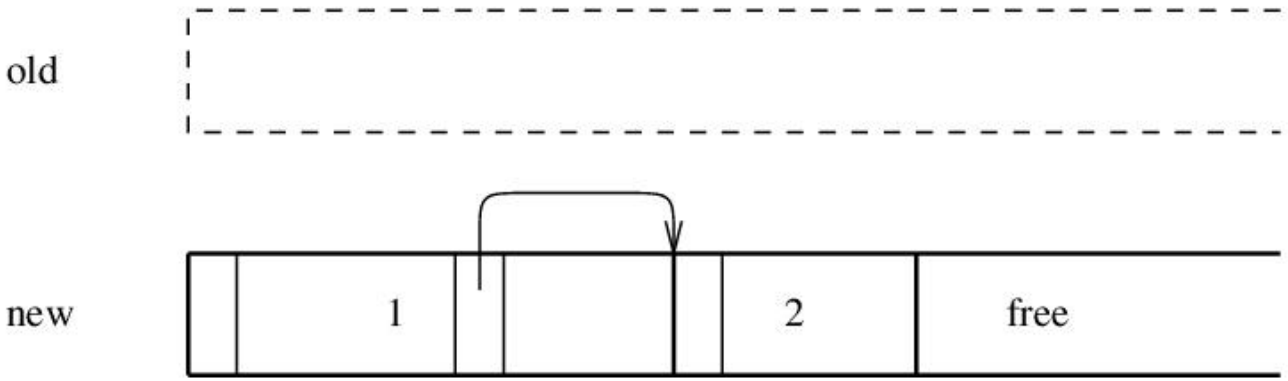

Figure 5.23 Moving the chunks during compaction.

Generational Garbage Collection

- Idee:
  - je länger ein Chunk besetzt ist, umso unwahrscheinlicher ist, dass er demnächst Müll wird
  - teste nur kürzlich allokierte Chunks auf Erreichbarkeit
    (die nach der letzten Kompaktifizierung allokierten)
  - dynamische Zweiteilung des Heaps
    (Chunks älterer und Chunks jüngerer Generation)
- Durchführung:
  - betrachte die ältere Generation als Programmdatenbereich
  - wende eine Heapbereinigung auf die jüngere Generation an
  - gelegentlich: bereinige den gesamten Heap
- Varianten:
  - Anzahl der Generationen
  - Abgrenzung zwischen alt und jung
  - genauer Kollektor-Algorithmus
  - Auffindung der Zeiger auf junge Chunks