Linz 2005

Reference Counting

Loidl Stefan (0255886 / 521)
Negeli Thomas (0255604 / 521)
Studenten der Johannes Kepler Universität Linz, Österreich
Papier für: Seminar Garbage Collection WS2005 339.372
StefanLoidl@gmx.net
ThomasNegeli@gmx.at

1. Einleitung

Wir zählen, wie oft eine Speicherzelle von verschiedenen Stellen im Programm aus verwendet wird. Dies bedeutet Reference Counting wenn man es auf den Punkt bringt. Viele Wissenschaftler haben sich mit dem Thema der Erstellung von effizienten Algorithmen beschäftigt.
Dieses Papier soll eine kompakte Zusammenfassung der gegebenen Literatur darstellen. Wichtige Algorithmen und Beispiele werden gebracht, die dem Verständnis der Zusammenhänge dienen sollen.
Kapitel 2 liefert eine Einführung in den Grundgedanken des Reference Counting.
Kapitel 3 geht auf detaillierterer Ebene an das Thema heran, um Speziallösungen und Optimierungen aufzuzeigen.
Kapitel 4 zeigt einige Grundgedanken zur Optimierung der Listenverwaltung auf.
Kapitel 5 beschäftigt sich mit dem grundlegendsten Problem aller Reference Counting Techniken; Zyklen in den Zellstrukturen erfordern spezielle Algorithmen um diese freizugeben.

2. Der Algorithmus, ein genereller Überblick

Freie Zellen haben einen Reference Count von 0, was man auch als Invariante bezeichnet.
Jedes mal, wenn eine neue Speicherzelle angefordert wird, wird dessen Reference Count auf 1 gesetzt. Wenn ein Zeiger auf diese Speicherstelle gesetzt wird, egal ob vom Heap oder vom Stack aus, wird der Referenzzähler der Zelle inkrementiert. Dekrementiert wenn ein Zeiger auf diese Zelle gelöscht wird. Wenn nun dieser Referenzzähler den Wert 0 erreicht, wissen wir, dass diese Zelle nicht mehr benutzt wird, und können diese freigeben.
Werfen wir nun einen detaillierten Blick auf Algorithmus 1, der uns die Anforderung von neuem Speicher verdeutlicht.
`free_list` verweist auf eine Menge von freien Speicherzellen, üblicherweise implementiert als verkettete Liste.
`newCell` ist eine einfache freie Speicherzelle.

Die Methode `allocate()` liefert uns eine neue freie Speicherzelle aus dem Heap. Nimmt also das erste Element aus der Liste, setzt den Zeiger für das nächste Listenelement um und gibt die freie Zelle zurück.
Die Methode `New()` kümmert sich dabei um die Verwaltung des Referenzzählers der neu angeforderten Zelle, und ob überhaupt genügend freier Speicherplatz zur Verfügung steht.
In Fall von Algorithmus 1 wird bei zu wenig Speicher abgebrochen, alternativ dazu könnte man den Heap erweitern.

```
/** Speicherzelle anlegen */
MemoryCell New() {
   if (free_list == null)
      /** Abbruch, kein Speicher
       * mehr frei */
         System.exit(1);
   newCell = allocate();
   newCell.rc = 1; //ReferenceCount
   return newCell;
}

/** Speicher anfordern */
MemoryCell allocate() {
   newCell = free_list.pop();
   free_list = free_list.next;
   return newCell;
}
```

Algorithmus 1: Anfordern von Speicher

Ein Update einer Speicherzelle erfordert ein bisschen mehr Arbeit. Algorithmus 2 verdeutlicht uns ein Überschreiben der Speicherzelle r mit der Speicherzelle s in der Methode Update.
Durch das Überschreiben des Speicherbereichs von r, wird natürlich der Speicher den das Objekt r zuvor belegt hat freigegeben. Also wird zuerst ein `delete(r)` aufgerufen.
Die Anzahl der Referenzen auf s steigt da ja nun auch der Speicherbereich von r darauf verweist, also `s.rc++`.
Anschließend erfolgt noch die Zuweisung von s auf r.

delete(…) dient dem Löschen der Speicherzelle. Wenn die zu löschende Speicherzelle keine weiteren Verweise mehr hat (*t.rc==0*) und t auf weitere Zellen verweist (*t.next!=null*) müssen auch diese rekursiv gelöscht werden.
Anschließend erfolgt mit *free(…)* eine Freigabe des Speichers, wobei die freizugebende Zelle an den Anfang der List der freien Zellen gespeichert wird.

```
/** Update von Objekten */
void Update(MemoryCell r,
            MemoryCell s) {
   delete(r);
   s.rc++;
   r = s;
}

/** Löschen von Referenzen */
void delete(MemoryCell t) {
   t.rc--;
      if(t.rc == 0) {
         if (t.next!=null)
            delete(t);
         free(t);
      }
}

/** Speicher freigeben */
void free(MemoryCell t) {
   t.next = free_list;
   free_list = t;
}
```

Algorithmus 2: Überschreiben von Zellen

2.1. Stärken und Schwächen des Reference Counting

Ein wesentlicher Vorteil dieser Art der Speicherverwaltung gegenüber anderen Techniken, ist die einfache Handhabung der Algorithmen. Speicherverwaltung und Programmausführung laufen verschränkt ab, im Gegensatz zu mark-and-sweep, wo die Bearbeitung der Programme während der Speicherverwaltung ausgesetzt wird.
Reference Counting bietet sich also z.B. bei Echtzeitsystemen an, wo ein Minimum an Antwortzeit gefordert ist.
Ein Problem beim vorgestellten Algorithmus 2 ist jedoch, dass in der Methode *delete(…)* rekursiv alle Kinder der Zelle gelöscht werden. Da Rekursivität sowohl in Bezug auf Verarbeitungsgeschwindigkeit als auch in Bezug auf Speicheraufwand nicht ideal ist, müssen andere Verwaltungsalgorithmen gefunden werden, die effizienter arbeiten.

Neben der einfachen Handhabung, ist ein weiterer Vorteil des Reference Counting die Räumliche Lokalität der Referenzen, die in den meisten Fällen nicht schlechter ist als die der laufenden Anwendung. Eine Zelle deren Reference Count einen Wert von 0 erreicht, kann ohne den Zugriff auf andere Zellen des Heaps, mit Ausnahme des bereits erwähnten Löschens der Kinder, zurückgefordert werden. Natürlich kann in Algorithmus 2 in der Update Methode ein Fehler auftreten, wenn eine der beiden verwendeten Zellen sich nicht im Hauptspeicher befindet.
Studien haben gezeigt [Clark77; Stoye84; Zorn89] [Hayes91] [Appel92] [BarrettZorn93b], dass wenige Zellen shared sind, viele jedoch eine kurze Lebenszeit haben. Unsere beiden Algorithmen Algorithmus 1 und Algorithmus 2 erlauben es, die freien Zellen sofort wieder zu verwenden, ähnlich einer Stack Implementierung. Anders verhält es sich bei Verfolgungsstrategien, oder auch Tracing Strategies bezeichnet, bei denen die freien Speicherzellen unbenutzt im Speicher verbleiben, bis der Garbage Collector sie wieder für die Verwendung freigibt. Die sofortige Wiederverwendung von Zellen erzeugt weniger Seitenfehler in einem System mit virtuellem Speicher und meist ein besseres Cache Verhalten als Verfahren mit Verfolgungsstrategien, solange der ganze Heap im Hauptspeicher gehalten werden kann. Sofortiges Wissen über frei werdende Zellen kann einen anderen möglichen Vorteil einbringen, nämlich genau dann, wenn eine leicht abgeänderte Kopie eines Objektes das frei geworden ist benötigt wird. Dann kann man sich einfach den Zeiger auf das Objekt „ausleihen“ und nur die Daten verändern, die nicht korrekt sind. Die Alternative dazu wäre, neuen Speicher zu allokieren, die Daten Wort für Wort aus der alten Zelle zu kopieren, und dann die alte Zelle zu löschen. Also unnötiger Arbeitsaufwand wenn man die Algorithmen „intelligent“ gestaltet. Diese Technik wird z.B. beim Glasgow Haskell compiler verwendet. [PeytonJones92]
Reference Counting kann auch Aufräumarbeiten oder „finalisation actions“ vereinfachen, z.B. beim Schließen von Dateien, wenn man den „Finaliser“ sofort beim Sterben eines Objektes aufruft.
Kein Vorteil ohne Nachteil, und auch das Reference Counting ist dagegen nicht gefeit.
Ein großer Nachteil ist der relativ hohe Berechnungsaufwand zur Aufrechterhaltung der Invariante. Jedes mal, wenn ein Zeiger überschrieben wird, müssen in der alten und in der neuen Zelle die Werte für den Reference Count justiert werden. Bei einer einfachen Tracing Strategie ist das nicht nötig, wenn auch der Verarbeitungsaufwand zu einem späteren Zeitpunkt nicht gerade minimal ist.
RC ist stark mit dem Speichermanagement des zugehörigen Benutzerprogrammes verbunden, bzw. mit dessen Compiler. Ein Update oder ein Kopiervorgang

eines Zeigers führt also in jedem Fall, zu einer Justierung der RC Felder der Zellen. Bei der Übergabe an Unterprogramme, müssen diese Zellen also inkrementiert, bei der Rückgabe dekrementiert werden. Diese Vorgehensweise darf kein einziges mal unterlassen werden. Man kann also sagen, dass RC eine sehr zerbrechliche Strategie ist, die durch die starke Kopplung mit dem Programm einen höheren Arbeitsaufwand mit sich bringt.
Des Weiteren muss in jeder Speicherzelle extra Speicherplatz gehalten werden, um den Wert des RC zu speichern. Im Schlimmsten Fall, auch wenn praktisch nicht relevant, muss dieses Feld fähig sein, die Gesamtzahl aller möglichen Zeigern zu speichern. Es muss also dieselbe Größe wie einer regulärer Zeiger haben.
Die größte Schwäche des RC ist die Unfähigkeit mit zyklischen Datenstrukturen umzugehen, z.B. mit doppelt verketteten Listen oder mit Bäumen, deren Blätter auf die Wurzel verweisen.
Abbildung 1 [Jones96] verdeutlicht diesen Sachverhalt anhand einer doppelt verketteten Liste.
Jedes Rechteck (R, S, T und U) stellt eine Speicherzelle dar, die Zahl n steht für den Reference Count Wert der Zelle.

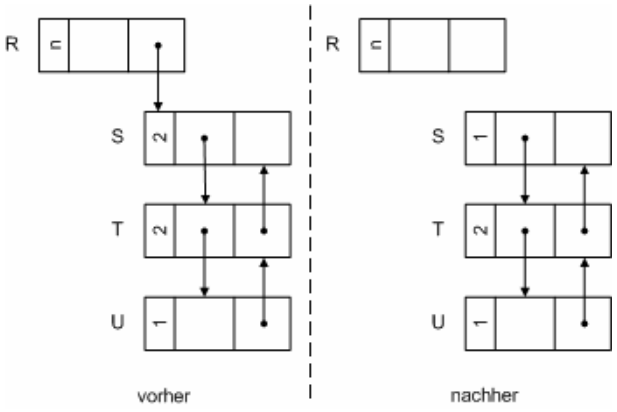


Abbildung 1: Problem mit zyklischen Strukturen

Vor dem Löschvorgang enthält die oberste Zelle n Referenzen. Auf S und T wird zweimal verwiesen, auf U nur einmal.
Nun löschen wir den Zeiger von R nach S und dekrementieren somit den RC der Zelle S. Die Teilliste S, T und U ist zwar jetzt nicht mehr zugänglich, kann aber nicht gelöscht werden, da S einen RC von 1 hat. Die Zellen bleiben somit im Speicher, können nicht genutzt, aber auch nicht freigegeben werden und somit ist ein Teil des Heaps verloren.
Glücklicherweise gibt es Techniken, die auch derartige Datenstrukturen handhaben können, und es empfiehlt sich sogar die Kombination von Reference Counting und Tracing Strategien [Weizenbaum69; Knuth73; DeutschBobrow76; Wise79].
Reference Counting wird so lange verwendet, bis der Heap voll ist. Anschließend wird eine Tracing Strategie verwendet. Der RC Wert jeder Zelle wird auf 0 gesetzt. Der Zähler jeder aktiven Zelle wird um 1 inkrementiert, jedes mal wenn sie innerhalb der Markierungsphase besucht wird. Da der Markierer jede Zelle genau einmal ausgehend von jedem Zeiger besucht, wird der korrekte Wert jeder Zelle wiederhergestellt.
Daraus ergeben sich wieder 2 Vorteile. Neben der Handhabung von zyklischen Datenstrukturen, können nun auch kleinere RC Felder verwendet werden. Zähler die den maximalen Wert des Feldes erreichen, werden von keinem Update mehr betroffen, stattdessen übernimmt die Tracing Strategie deren Verwaltung.
Eine ähnliche Vorgehensweise bietet sich auch für die Verwaltung von Listen an. Die konkrete Umsetzung ist in [McCarthy60] nachzulesen, bzw. eine kurze Zusammenfassung findet sich in [Collins60].

3. Nicht rekursives Löschen

In den einführenden Algorithmen Algorithmus 1 und Algorithmus 2, dekrementiert ein Update den RC einer Zelle, wenn diese überschrieben wird. Erhält dadurch der RC den Wert 0, werden rekursiv alle Kinder der Zelle gelöscht, noch bevor der zugehörige Speicher freigegeben wird.
Der Aufwand des Löschens des letzten Zeigers auf eine Zelle ist nicht konstant, und auch nicht proportional zur Größe des Objektes, jedoch abhängig von der Größe des Sub-Graphen, also von der Anzahl der Kinder einer Zelle. Dieser Sachverhalt stellt ein Problem dar, das zu bewältigen ist.

3.1. Weizenbaum's Algorithmus (lazy freeing algorithm)

Weizenbaum's Vorgehensweise [Weizenbaum63] birgt eine minimale Abänderung der Algorithmen Algorithmus 1 und Algorithmus 2 in den Methoden `New()` und `delete(…)`.
Die Position des rekursiven Löschens der Kinder hat sich geändert. Die Weizenbaum Vorgehensweise ist also eine völlig andere. Anstatt die Kinder einer nicht mehr referenzierten Zelle gleich bei der Freigabe der Zelle zu löschen, wird die Zelle zunächst in die Liste der Freien Speicherzellen gelegt, die als Stack implementiert ist. Der Inhalt der Zellen wird dabei nicht verändert, sodass weitere Zeiger auf andere Zellen unverändert verbleiben. Wenn diese Zelle mittels *New()* wieder aus dem Stack genommen wird, werden ihre Kinder rekursiv gelöscht.
Zur Verkettung der Elemente im Stack der freien Zellen, kann nun das Reference Count Feld verwendet werden,

da garantiert ist dass es ohnehin 0 ist wenn keine Referenzen auf das Objekt bestehen.

```
/** Speicherzelle anlegen */
MemoryCell New() {
   if (free_list == null)
      /** Abbruch, kein Speicher
       * mehr frei */
      System.exit(1);
   newCell = allocate();
   if(newCell.next != null)
      delete(newCell);
   newCell.rc = 1; //ReferenceCount
   return newCell;
}
```

Algorithmus 3: Lazy Freeing

Spätestens hier wird deutlich, dass dieses Feld nur ein Zeiger sein kann.

```
delete(t) =
   if (t.rc==1) {
      t.rc = free_list
      free_list = t
   }
   else decrementRC(t)
```

Algorithmus 4: Freigabe einer Zelle

t.rc==1 (Algorithmus 4) sagt uns jetzt dass die Zelle *t* von keinem anderen Objekt als vom aktuellen mehr referenziert wird, also machen wir *t* zum ersten Element im Stack der freien Zellen.
Zusätzlich abstrahieren wir das Dekrementieren des Reference Zählers in die Methode *decrementRC(…)*.

3.1.1. Kosten und Nutzen dieser „faulen“ Implementierung

Die relativ geringe Differenz zur ursprünglichen Vorgehensweise in Algorithmus 1 und Algorithmus 2 [Jones96] bringt uns in einem Punkt einen Vorteil. Im Falle von kaskadierten Freigaben von Zellen. Leider löst sie das Problem der ungleichen Bearbeitung nicht vollständig, denn wenn z.B. ein Array an erster Postition im Stack liegt, und diese Zelle angefordert wird, ist die Bearbeitungszeit zur Freigabe des Speichers noch immer von der Größe des Arrays abhängig.
Die Faulheit in Algorithmus 3 kann auch zu einer Verschwendung von Speicherplatz führen, wie sie z.B. zuvor bei zyklischen Datenstrukturen erwähnt wurde. Die Zellen die eine Datenstruktur verbraucht bleiben nach der Freigabe unbrauchbar so lange sie nicht mittels *New()* freigegeben und neu zugewiesen werden. Nehmen wir also als Beispiel eine Datenstruktur mit einem kleinen Header, der auf einen großen, speicherhungrigen Body zeigt. Wenn diese Datenstruktur gelöscht wird, kommt der Header, der wenig Speicher verbraucht, auf den Stack mit den freien Zellen. Anschließend löschen wir einige Zeiger auf kleinere Objekte die wiederum an die Spitze des Stack gelegt werden. Somit ist die Speicherverschwendung durch den unnützen Body längere Zeit vorhanden.

3.2. Aufschiebetaktik

Der Verwaltungsaufwand zur Handhabung von Reference Counting kann also hoch sein, wodurch diese Technik weniger attraktiv als Verfolgungsstrategien zu sein scheint [Hartel88]. Das Überschreiben von Zeigern erfordert einiges an Arbeitsaufwand um die Count Werte in alter und neuer Zelle zu justieren. Ähnliches geschieht, wenn man einen Zeiger vom oder auf den System Stack gibt. Sogar nicht destruktive Operationen wie Unterprogrammaufrufe oder das Durchwandern einer Liste erfordern die Manipulation der Count Werte. Ein Inkrement wenn eine Zelle übergeben oder besucht wird, und ein Dekrement wenn zurückgesprungen wird. In Architekturen mit Caches führt Reference Counting oft dazu, dass Daten in den Cache geladen werden, die ansonsten gar nicht berührt worden wären. Diese Daten werden verändert und müssen deshalb wieder in den Speicher zurückgeschrieben werden, obwohl sie wieder den Ursprungswert wie vor dem Aufruf haben [Baker94].
Es ist also offensichtlich, dass eine Veränderung der Count Werte nach Möglichkeit vermieden werden sollte. Eine mögliche manuelle Technik ist eine Sonderbehandlung bei Unterprogrammaufrufen. Wenn also garantiert ist, dass ein Count Wert nicht null wird, kann die Manipulation der Werte unterlassen werden. Wissen über ein Unterprogramm wird also impliziert, was im manuellen Modus, also vom Benutzer selbst gehandhabt, kein Problem ist. Diesen Vorgang will man natürlich automatisieren, also überlässt man diese Optimierungsarbeit dem Compiler.

3.2.1. Deutsch-Bobrow Algorithmus

Deutsch und Bobrow [DeutschBobrow76] haben eine Vorgehenweise zur Speicherverwaltung zur Programmlaufzeit entwickelt. Diese Technik erfordert also keine Anpassung der Compiler.
Studien haben gezeigt, dass der Hauptteil an Speicheroperationen von Zeigern in lokalen Variablen gemacht wird, also im System Stack. Die Häufigkeit von anderen Operationen liegt bei modernen Compilern im Bereich von 1%. [Taylor86; Appel89; Zorn89]

Diesen Sachverhalt nutzen Deutsch und Bobrow dahingehend aus, dass lokale Variablen speziell behandelt werden, indem für sie keine Reference Counts bei Modifikationen geführt werden. Schreibvorgäng von Zeigern bei lokalen Variabeln können somit durch einfache Zuweisungen ersetzt werden, ein Update in der urspünglich komplexen Form ist nicht mehr nötig.

```
/** Update von Objekten */
void Update(MemoryCell r,
            MemoryCell s) {
    delete(r);
    s.rc++;
    ZCT.remove(s);
    r = s;
}

/** Löschen von Referenzen */
void delete(MemoryCell t) {
    t.decrementRC();
    if (t.rc==0) {
        ZCT.put(t);
    }
}
```

Algorithmus 5: Überschreiben von Zellen bei Deutsch-Bobrow

Count Werte spiegeln jetzt also nur mehr die Referenzen von andern Heap Objekten wieder, somit können Zellen nicht mehr freigegeben werden wenn ihr Count Wert 0 erreicht, denn dann sind sie eventuell noch immer von lokalen Variabeln erreichbar. Stattdessen werden Zellen mit Count Werten von 0 in der *Delete(…)* Methode in eine Hashtabelle bzw. eine Bitmap (Erklärung in Kapitel 3.2.2) geschrieben, die Zero Count Table oder ZCT bezeichnet wird. Algorithmus 5 [Jones96] verdeutlicht diese Vorgehensweise.
Eine neue Referenz im Heap führt zum Löschen des Elements aus der Hashtabelle und zu einem Inkrement des Count Wertes des Elements.
Anschließend wird diese Tabelle zyklisch in 3 Phasen durchsucht:
Alle Elemente der ZCT mit Referenzen aus dem Stack werden markiert.
Alle nicht markierten Elemente werden entfernt.
Alle Markierungen werden entfernt.
Eine Möglichkeit Elemente zu markieren bzw. die Marken wieder zu entfernen ist Algorithmus 6 zu entnehmen.
Ein Objekt in der ZCT kann nur dann Müll sein, wenn es nachdem alle Elemente die vom Stack aus direkt zugreifbar sind ihren Count Wert erhöht haben, noch immer *rc==0* hat. Anschließend löschen wir diese Elemente inklusive ihrer enthaltenen Zeiger, und dekrementieren die Counts der per Stack referenzierten Zellen wieder. Wir stellen somit den Ausgangszustand wieder her.
Da der Stack üblicherweise wesentlich kleiner ist als der Heap, bringt uns dass zum einen beim Freigeben der Objekte durch das Programm Geschwindigkeit, zum anderen bei der Verwaltung des Speichers durch das System.

```
void reconcile() {
  /** markieren der Elemente */
  for(i=0; i<stack.getSize(); i++) {
     stack.getElem(i).incrementRC();
  }

  /** unreferenzierte Zellen
   * löschen */
  for(i=0; i<ZCT.getSize(); i++) {
     cell = ZCT.getElement(i);
     if (cell.rc == 0) {
        if (cell.next!=null)
           delete(cell);
        free(cell);
     }
  }

  /** Markierungen entfernen */
  for(i=0; i<stack.getSize(); i++) {
     stack.getElem(i).decrementRC();
  }
}
```

Algorithmus 6: Aufräumen des Heap

Sehen wir uns hierzu ein Beispiel an, die Berechnung des Größten gemeinsamen Teilers 2er nicht negativer Zahlen, Algorithmus 7 [Jones96].
Folgende Konventionen sind für das Beispiel getroffen
Alle Objekte werden am Heap abgelegt
Ausdrücke werden als Graph dargestellt dessen Knoten Objekte am Heap sind.
Der System Stack enthält Zeiger auf Heap Objekte.
Atomare Objekte werden mit ihrem Wert bezeichnet.

```
/** Größter gemeinsamer Teiler
 * Bedingung: x >= y >= 0 */
int gcd(int x, int y) {
    if (y == 0)
        return x;
    t = x-y;
    if (x>t)
        return gcd(y,t);
    else
        return gcd(t,y);
}
```

Algorithmus 7: Größter gemeinsamer Teiler mittels Deutsch-Bobrow

Wir füttern Algorithmus 7 mit den Werten 18 und 12, also `gcd(18,12)`. Eine Auswertung per Hand würde im ersten Schritt zu einem Aufruf von `gcd(12,6)` führen, sehen wir uns an, wie das System dies erledigt.
Als erstes wird der Graph erzeugt.
R (eine Datenstruktur, die einen Reference Count, einen right und einen left Zeiger speichern kann), die beiden Argumente und ein Zeiger auf die Funktion gcd werden im Stack abgelegt, bzw. ein Zeiger auf diese Heap Objekte laut Konvention. Alle Knoten haben einen Reference Count von 0, außer dem Knoten R der möglicherweise geshared ist, und die ZCT ist leer, wie in Abbildung 2 ersichtlich.

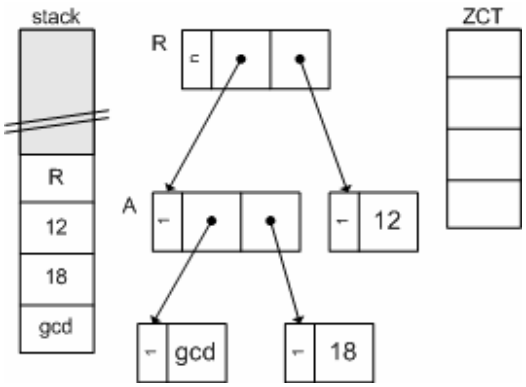


Abbildung 2

`t` wird mit 6 belegt, da `y!=0` gilt, also wird eine neue Speicherzelle angefordert und mit dem Wert 6 belegt. Da auf dieses Element vom Heap aus nicht referenziert wird, erhält es einen Count von 0 und wird der ZCT hinzugefügt.

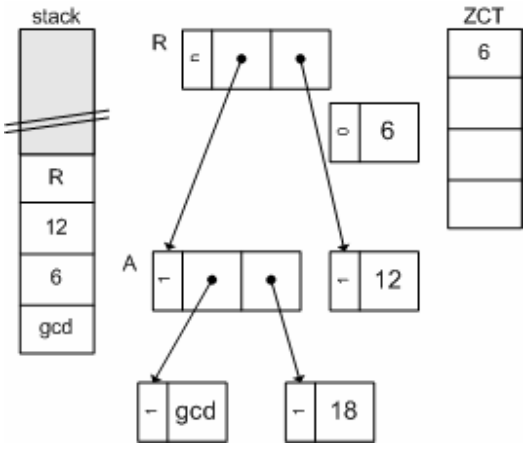


Abbildung 3

Der Compiler erkennt dass 18, unser x Argument, nicht mehr benötigt wird, und verwendet dessen Slot im Stack für den Zeiger auf die 6, womit dann auch verhindert wird, dass die 6 beim Anstoßen des `reconsile()` wieder aus dem Heap gelöscht wird. Diesen Zustand des Speichers sehen wir in Abbildung 3.
Als nächstes binden wir 6 mittels `Update(right(R),6)` in den Graphen ein, erzeugen eine neue Zelle B und führen wieder ein `Update(left(R),B)` durch.

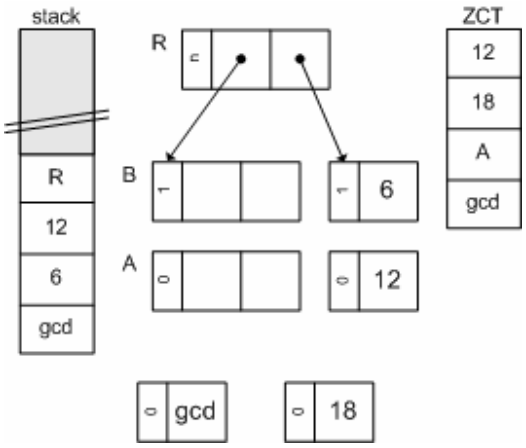


Abbildung 4

Die Einbindung von 6 in den Graphen erhöht den RC von 6 und bewirkt ein Löschen von 6 aus der ZCT. Da `left(R)` überschrieben wird, werden die Zeiger auf A, gcd und 12 rekursiv gelöscht. Zu diesem Zeitpunkt enthält die ZCT die Objekte 12, 18, A und gcd, wie in Abbildung 4 ersichtlich.
Durch das Befüllen der ZCT, wird ein `reconcile()` angestoßen (siehe Algorithmus 6 [Jones96]). Beim Durchlauf des Stacks werden R, 6, 12 und gcd gefunden und markiert, also ihr RC inkrementiert. A und 18 bleiben bei einem RC von 0, werden aus der ZCT entfernt und in die Liste der freien Zellen eingehängt. 12 und gcd werden vom Stack aus referenziert und bleiben somit in der ZCT. Siehe Abbildung 5.

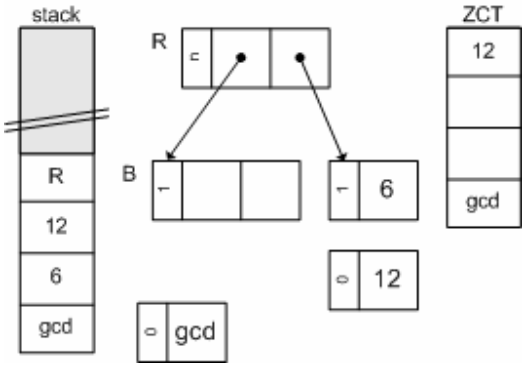


Abbildung 5

Als nächsten Schritt würde das System B mit gcd und 12 verlinken (Abbildung 6 [Jones96]) und das Spiel beginnt von Vorne.

3.2.2. ZCT Überlauf

Ein Nachteil der Vorgehensweise aus Kapitel 3.2.1 ist, dass der Versöhnungsvorgang, das `reconcile()`, erst beim Überlauf der ZCT angestoßen wird, obwohl möglicherweise mehrere Elemente in die ZCT eingefügt werden wenn ein Element freigegeben wird (wenn die Freigabe einer Zelle die Freigabe mehrere Zellen nach sich zieht).

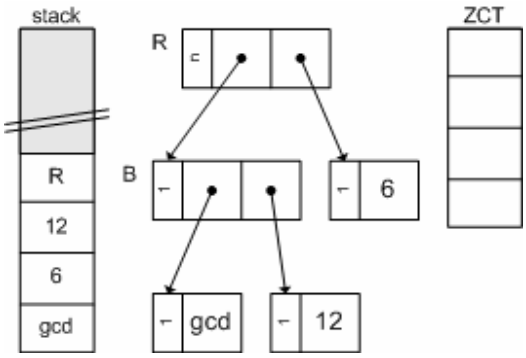


Abbildung 6

Für dieses Problem gibt es verschiedenste Lösungen.
Verursacht die Freigabe einer Zelle einen Überlauf, so kann deren Behandlung bis zum nächsten `reconcile()` ausgesetzt werden.
Alternativ dazu kann Weizenbaum's Algorithmus angepasst werden, bei dem die Zeiger eines freigegebenen Objektes erst bei der erneuten Anforderung der Zelle gelöscht werden. Wenn dann die ZCT überläuft, wird ein `reconcile()` angestoßen.

	Immediate Freeing	Deferred Freeing
Updates	15	3
Reconciliation		3
Recusive Freeing	5	5
Total	20	11

Abbildung 7: Gewinn der Aufschiebetaktik gegenüber sofortiger Freigabe

Wird die ZCT als Bitmap implementiert, stellt ein Überlauf kein Problem dar [Baden83]. Bei der Garbage Collection stellt ein Bitmap ein Array von Bits dar, wobei jedes Bit ein Wort aus dem Heap repräsentiert. Ein setzen bzw. löschen eines Bits steht dann für das einfügen bzw. löschen eines Elements der ZCT.

3.2.3. Effizienz der Aufschiebetaktik

Sehr effizient ist dieses Verfahren bei der Reduzierung der Kosten für den Schreibvorgang eines Zeigers.
Abbildung zeigt den Gewinn der Deferred Freigabe (Aufschiebetaktik) gegenüber der Immediate Freigabe. Die Werte repräsentieren Prozente der absoluten Ausführungszeit und stammen aus Versuchen mit der Smalltalk Sprache aus den 80er Jahren.
Der Hauptnachteil jedoch liegt im Verlust des durch das Reference Counting gewonnen Vorteils der sofortigen Freigabe von Speicher der nicht mehr benötigt wird. Dieser Speicher wird erst bei einem `reconcile()` freigegeben, und nicht sofort nach freiwerden der Zelle.

3.3. Reference Counting mit beschränkten Feldgrößen

Wie bereits in Kapitel 2.1 erwähnt, benötigt Reference Counting in jeder Zelle ein eigenes Feld zur Speicherung der Anzahl an Referenzen auf diese Zelle. Dieses Feld ist im schlimmsten Fall so groß wie die maximale Anzahl an Zeigern von Heap und Stack, also so groß wie ein Zeiger in der jeweiligen Architektur.
Die Praxis zeigt dass es für eine Anwendung unmöglich ist, derartig hohe Zählerständen zu erreichen, also kann in diesem Feld Speicher eingespart werden, wenn man die Behandlung von Überläufen spezialisiert.

3.3.1. Klebrige Zähl-Felder

Kleine Felder für die Reference Counts können zu einem Überlauf führen, wenn mehr Referenzen exisiteren als gespeichert werden können. Daraus folgen also 2 Probleme:

```
/** Referenzzähler Dekrement mit
 * Sticky Wert */
void decrementRC() {
   if (this.rc < sticky)
      this.rc--;
}

/** Referenzzähler Inkrement mit
 * Sticky Wert */
void incrementRC() {
   if (this.rc < sticky)
      this.rc++;
}
```

Algorithmus 8: Operationen auf Reference Counts mit Sticky Werten

Ein Zählerstand darf seinen maximal zulässigen Wert nicht überschreiten, und sobald ein Zähler seinen maximalen Wert annimmt, bleibt er fix auf diesem „Sticky“ Wert stehen, siehe Algorithmus 8 [Jones96].
Er darf nicht reduziert werden solange mehr Referenzen exisitieren als der maximale Wert anzeigt.

3.3.2. Reference Counting alleine ist nicht genug

Aus den in Kapitel 3.3.1 genannten Problemen folgt, dass eine Zelle deren Count Wert einmal den sticky Wert erreicht hat, nie mehr freigegeben werden kann, wenn nur Reference Counting eingesetzt wird.
Deshalb wird zusätzlich eine Verfolgungsstrategie eingesetzt, praktisch als Rückendeckung für das Reference Counting, um die echten Count Werte wieder herzustellen. Wir durchwandern den Heap und setzen alle Reference Counts auf 0. Anschließend markieren wir alle aktiven Zellen indem wir deren RC inkrementieren. Jede Zelle erreicht somit ihren ursprünglichen Wert an Referenzen bzw. den „sticky“ Wert, je nachdem welcher kleiner ist (Siehe Algorithmus 9 [Jones96]). Der Einsatz dieser Zusatzstrategie ist nicht lästig, da sie ohnehin nötig ist um zyklische Datenstrukturen zu behandeln, wie in Kapitel 2.1 bereits erwähnt.
Der Einfachheit halber stellen wir *mark()* rekursiv dar, in Wirklichkeit werden natürlich effizientere Techniken eingesetzt, hierzu möchte ich jedoch auf das Paper zum Thema Mark & Sweep von Kollegin Schartner verweisen.

3.3.3. Reference Counting mit einem einzelnen Bit

Ein weiterer radikaler Ansatz ist die Reduzierung des Count Feldes auf ein einzelnes Bit [FriedmanWise77; Stoye84; ChikayamaKimura87; Wise93]. Eine Zelle ist dann entweder „shared“, also „sticky“, oder nicht. Diese Vorgehensweise ist durch Studien belegt, in denen gezeigt wurde dass ein Großteil der Zellen nicht „shared“ sind, und somit sofort nach Löschen des betreffenden Zeigers freigegeben werden können. Ein Hautfokus des RC liegt also in der Verwaltung derartiger nicht „shared“ Zellen.
Die Ziele des 1 Bit Reference Counting sind, den Vorgang der Garbage Collection so lange wie möglich hinauszuzögern da dadurch eine Verzögerung der Benutzerprogrammausführung bewirkt wird, und den benötigten Speicherplatz der Speicherverwaltung selbst dramatisch zu reduzieren. Die Vermeidung unnötiger Kopien durch das Ausborgen von Zeigern und „in-place“ Updates wurden bereits in Abschnitt 2.1 erwähnt. Diese Vorgehensweise findet sofort Anklang wenn man an Arrays mit einigen Tausend Elementen denkt die kopiert oder manipuliert werden sollen.
Die einfachste Implementierung wäre also, das unique Bit in jeder Zelle zu speichern [FriedmanWise77]. Praktischer ist jedoch das Speichern des Bits in jedem Zeiger [Stoye84], als tagging bezeichnet.

```
void mark_sweep() {
   /** löschen aller Count Werte */
   for(i=0; i<Heap.SIZE; i++) {
      Heap.getCell(i).rc = 0;
   }

   /** markieren aller
    * referenzierten Zellen */
   for(i=0; i<Stack.SIZE; i++) {
      mark(Stack.getCell(i));
   }

   /** bereinigen des Speichers */
   sweep();
   if (free_list==null)
      /** Abbruch, kein Speicher
       * mehr frei */
      System.exit(1);
}

void mark(MemoryCell mc) {
   mc.incrementRC();
   if(mc.rc==1) {
      /** rekursiv alle Kinder
       * markieren */
      if(mc.next!=null)
         mark(mc.next);
   }
}
```

Algorithmus 9: einfacher Mark & Sweep

Der erste Zeiger auf ein neu erzeugtes Objekt wird mit dem Tag unique belegt. Wird ein Zeiger kopiert, so erhält er den Tag sticky, wie es z.B. bei einem Update geschieht (siehe Algorithmus 10). Zusätzlich wird noch der Quellpointer auf seine unique Bit geprüft, denn dann muss es ebenfalls auf sticky gesetzt werden.

```
Update(R,S) =
   delete(*R)
   T = sticky(*S)
   if RC(*S) == unique
      *S = T
   *R = T
```

Algorithmus 10: Überschreiben von Zellen mit Sticky Zeigern

Durch das Speichern des Bits im Zeiger selbst, kann das Auslesen und Modifizieren des Tags erledigt werden, ohne den Inhalt der Zelle auszulesen. Dadurch wird die Wahrscheinlichkeit eines Cache Zugriffsfehlers oder

eines Seitenfehlers verringert werden. In Relation zum Zeitverlust durch einen Cache Miss oder Page Fault, ist der Aufwand durch die zusätzlichen Operationen im Update leicht zu verkraften.

3.3.4. Wiederherstellen des unique Zustandes

Sticky Zeiger können mittels Reference Counting nie mehr zu unique Zeigern werden, dafür ist wie bereits erwähnt ein Tracing Algorithmus nötig.
Rücken wir das Reference Count Bit vom Zeiger in den Knoten selbst, also in die Zelle, können wir das zugehörige Feld mit dem mark Bit der *mark_sweep()* Methode aus Algorithmus 9 [Jones96] teilen. „sticky“ wird dann mit markiert gleichgesetzt. Nachdem alle referenzierten Zellen markiert wurden, verbleiben also alle Zellen im „sticky“ Status. Des Weiteren kann nicht mehr zwischen shared und einmal referenziert unterschieden werden.
Im Moment haben wir auch keine direkte Möglichkeiten den unique Zustand einer Zelle wieder herzustellen. Es gibt jedoch Lösungen für dieses Problem, näheres hierzu ist ebenfalls beim Thema Mark & Sweep nachzulesen.
Die Markierung der einzelnen Zellen als unique oder sticky, hat in der Verwaltung von Listen ihr Äquivalent in den Bezeichnungen „owned“ und „borrowed“, siehe dazu [Gelernter60] bzw. [Collins60].

3.3.5. Der „Ought to be Two“ Cache

Viele Veränderungen der Count Werte sind nur vorübergehend.
Nehmen wir z.B. die Anweisung *N = select(N)*, wobei N einen Count Wert von unique hat, und select eine Projektionsfunktion auf ein bestimmtes Feld von N darstellt, z.B. das letzte Element einer Liste. Der Count wert der Zelle *select(N)* muss auf sticky gesetzt werden, noch bevor dir Referenz auf N aufgelöst wird, ansonsten würde die Zelle freigegeben werden noch bevor ihr Inhalt ausgelesen werden konnte. Somit ist wiederum die unique Information verloren.
Eine Lösung hierfür ist ein Software Cache mit Elementen, deren Count Wert unique ist, aber in Wirklichkeit schon 2 sein sollte, deshalb auch der Name dieser Technik [FriedmanWise77], die in Algorithmus 11 verdeutlicht wird. Sobald ein Zeiger auf einen unique Knoten kopiert wird, wird der Knoten in den Cache eingefügt. Wenn er bereits im Cache ist (bezeichnet wird dies als hit), wird dieser wieder aus dem Cache entfernt und als shared bzw. sticky markiert. Ein Cache Überlauf wird dahingehend gehandhabt, als dass zufällig ein Element entfernt wird und dieses als „sticky“ markiert wird. Eine mögliche Strategie die dies nicht dem Zufall überlässt wäre z.B. least recently used.

```
/** Update von Objekten */
void Update(MemoryCell r,
            MemoryCell s){
    if(s.rc==unique)
        insert(s);
    delete(r);
    r = s;
}

/** Cache füllen */
void insert(MemoryCell s) {
    if(hit(s))
        /* markieren wenn bereits im
         * Cache */
        s.rc = sticky;
    else
        cache.put(s);
   }

/** Löschen von Referenzen */
void delete(MemoryCell t) {
    if(!hit(t)) {
        if(t.rc == unique) {
            if(t.next!=null)
                delete(t.next);
              free(t);
          }
      }
}

/** Überprüfen ob Zelle bereits im
* Cache ist */
boolean hit(MemoryCell s) {
    if(cache.contains(s)) {
       cache.remove(s);
       return true;
    }
    else return false;
}
```

Algorithmus 11: Update mit Ought to be Two Cache

Wenn ein Zeiger gelöscht wird, wird der Knoten wieder aus dem Cache genommen (wenn er denn im Cache existiert) und somit der unique Zustand wieder hergestellt. Befindet sich die Zelle nicht im Cache, ist aber unique, so wird die Zelle rekursiv freigegeben.
Natürlich ist diese Strategie nur dann sinnvoll, wenn der zugehörige Cache sehr schnell ist, was z.B. durch die Benutzung von einigen wenigen Registern gewährleistet wäre. Für unser Beispiel *N = select(N)* würde ein einzelnes Register genügen, um die Erhöhung der Count Werte zu unterbinden. Derartige Zuweisungen findet man oft wenn man eine verkettete Liste durchläuft.

3.4. Reference Counting per Hardware

Trotz all dieser Optimierungen ist es anerkannt, dass Reference Counting mehr Zeit in Anspruch nimmt als Verfolgungsstrategien, weshalb auch die Implementierung in Hardware nahe liegend ist. „self-managing heap memories based on reference counting“ wird diese Technik genannt [Wise85; Wise94; GehringerChang93]. Im Prinzip handelt es sich dabei um aktiven Speicher, entgegen der Von Neumann Architektur, die strikt Intelligenz, also die CPU, vom Speicher trennt. Jegliche buchhalterische Tätigkeit zur Handhabung von Reference Counts wird auf spezielle Bänke, die man Reference Counting Memory (RCM) nennt, übertragen. Die CPU bleibt also frei für nützliche Dinge. Neben der Entlastung der CPU ermöglicht diese Trennung auch den einfachen Einsatz von RC unter Multi Prozessor Systemen, ohne auf die Synchronisation von Nutzerprogrammen oder Verfolgungsstrategien achten zu müssen, oder etwa auf locks bei Count Feldern.
Architekturen mit speziellem Zweck haben jedoch in der Geschichte keinen wirklich hohen kommerziellen Erfolg gebracht, da sich die immens hohen Entwicklungskosten oft nicht rechnen. Wenn man jedoch der CPU eine völlig eigenständige einfache Speicherbank vorgaukelt, erhält man eine Architektur die in einem weiten Spektrum von Anwendungsfeldern eingesetzt werden kann, womit die Entwicklungskosten wieder verteilt getragen werden können.
In jeder dieser Speicherbank findet sich ein Bereich für einen Standard Datenspeicher und einen RCM Bereich. Jeder dieser Bereiche hat seinen eigenen Bus und seine eigenen Ports, einen Datenport zu den CPUs und einen schmaleren Port zu den andern RCM’s. Letzterer läuft mit der doppelten Geschwindigkeit des Datenports, da ein einfacher Daten Schreibbefehl schon 2 RC Operationen auslösen kann, ein inkrement und ein dekrement. Jede RCM Bank verwaltet seine eigene Liste an freiem Speicher.
Die Performance des Systems ist abhängig von der Problemgröße. Bei einem entsprechend großen Problem, haben Versuche eine Ausführungszeit von 40 bis 70% der Zeit auf regulärer Hardware mittels Softwareverwaltung ermittelt.

4. Cyclic Reference Counting

Die bisher vorgestellten Algorithmen optimieren Teilaspekte der Reference Counting Verfahrens. Auf das sicherlich gröbste Problem wurde allerdings noch nicht eingegangen; Zyklische Zellstrukturen können durch das Verfahren nicht aufgelöst werden [McBeth63], wodurch es zu einem Speicherverlust in der weiteren Berechnung kommt. Zyklen entstehen auf der Programmebene etwa bei der Verwendung einer doppelt verketteten Liste. Auf der Systemebene verwenden funktionale Programmiersprachen häufig Zyklen um Rekursion auszudrücken.
Zwei offensichtliche Möglichkeiten um das Problem zu umgehen sind einerseits Zyklen zu vermeiden, was zu einer Restriktion von Programm- und Programmierstiel führen würde. Andererseits könnte ein zweiter Garbage Collector dazu verwendet werden periodisch, zyklische Daten freizugeben. Wie eingangs bemerkt, umgeht man damit das Problem eher als das man es löst. Natürlich wurden aber auch Methoden entwickelt um das Problem der zyklischen Referenzen direkt anzugehen.
Einige davon werden in Folge vorgestellt, es sollte jedoch erwähnt werden, dass laut [Jones96] keine jener Methoden bisher für den Einsatz in einem signifikanten System adaptiert wurde.

4.1. Funktionale Programmiersprachen

Das Erzeugen von Referenzen auf Zyklen in funktionalen Programmiersprachen erfolgt nach einem relativ engem Schema [Friedman79]. Diese Tatsache erleichtert es Zyklen zu erkennen und ihnen eine spezielle Behandlung zukommen zu lassen.
Wie bereits angemerkt entstehen Zyklen in funktionalen Programmiersprachen ausschließlich durch rekursive Definitionen. Es genügt einige eingeschränkte Regeln zu beachten um diese zu erkennen [Jones96]:

- Ein ganzer Zyklus wird immer auf einmal erzeugt.
- Wird eine Untermenge des Zyklus verwendet die den Wurzelknoten nicht mit einbezieht, so wird diese in eine unabhängige Struktur kopiert. Es werden also keine Knoten gemeinsam verwendet.
- Pointer die den Zyklus zum Wurzelknoten schließen werden auch explizit als solche gekennzeichnet.

Durch das Einhalten dieser Regeln wird ein Zyklus auf einen einzelnen Ansatzpunkt reduziert. Um auf Knoten zugreifen zu können ist es nötig den Weg über den Wurzelknoten zu nehmen. Man kann den Zyklus also als geschlossenes Objekt betrachten, das über den Wurzelknoten erreichbar ist. Werden alle Verbindungen, außer den gekennzeichneten Internen, zu diesem Hauptknoten getrennt so ist keines der Elemente mehr erreichbar. Der ganze Zyklus kann also aus dem Speicher entfernt werden.

4.2. Bobrow’s Technik

Treibt man das Konzept der internen und externen Pointer das im Absatz über funktionale Programmiersprachen schon ansatzweise verwendet wurde noch etwas weiter, um es zu generalisieren und noch weiter auszureizen passiert folgendes. Man behandelt einen Zyklus wie ein eigenständiges Objekt. Innerhalb eines Zyklus gibt es nur interne Pointer die nicht gezählt werden. Referenzen die

von außen auf irgendeines der zum Zyklus gehörigen Objekte zeigen werden als externe Referenz auf den gesamten Zyklus gezählt. Der Zyklus als ganzes bleibt daher genau so lange bestehen bis alle Referenzen auf die Objekte des Zyklus getrennt wurden, danach kann er wie ein einzelnes Objekt freigegeben werden.
Bobrow setzte auf diesem Gedanken auf und entwickelte ihn zu Gruppierungen von Zellen weiter [Bobrow80]. Alle angelegten Zellen werden durch den Programmierer zu einer Gruppe zugeordnet. Referenzen werden nun nur mehr für die einzelnen Gruppen gezählt, um jedoch effizient zu bleiben ist es notwendig, dass sich die Gruppe eine Zelle einfach aus ihrer Adresse ableiten lässt, oder sich sonst irgendwie einfach und schnell ermitteln lässt. Bobrow unterscheidet grob in zonenbasiertes Referenzzählen, und internes Beschriften von Gruppen. Erstes entspricht dem Ansatz die Gruppennummer direkt aus der Adresse abzuleiten, hat damit jedoch den großen Nachteil, dass es zu Speicherverlust kommen kann falls eine Datenstruktur um vieles kleiner ist als eine Zone. Bei den beschriftenden Verfahren kann man weiter unterteilen in solche die einer Zelle eine Gruppennummer hinzufügen und solche die direkt eine Referenz auf die Referenzzahl der Gruppe speichern.
Algorithmus 12 zeigt ein Pseudocodestück für das Speichern eines Pointers in eine Zelle. Verwendet wird ein Verfahren bei dem die Gruppennummer bei der Erstellung fix vergeben wird und sich zu Laufzeit nicht ändern kann.
Alternativ dazu beschreibt Bobrow noch eine weitere Technik bei der sich die Gruppennummer zur Laufzeit verändern kann. Um dies zu erreichen ist es nötig, dass der Programmierer vorgeben kann ob ein Pointer rein intern ist. Eine Zelle kann dann erzeugt werden ohne anfangs bereits die Gruppenzugehörigkeit zu deklarieren. Diese wird erst festgelegt, wenn mit dem ersten interne Pointer auf die Zelle verwiesen wird, oder der erste interne Pointer zu einer anderen zyklischen Zelle in der Zelle abgelegt wird. Während der Zeit in der sich eine Zelle noch nicht in einer fixen Gruppe befindet muss dennoch die Anzahl ihrer Referenzen gespeichert werden. Das wird über eine temporäre Gruppennummer erreicht. Wird das Element schlussendlich in eine Gruppe eingefügt, so muss die Referenzanzahl meist nur zur Gruppenreferenzanzahl addiert werden. Ähnlich kann man auch vorgehen wenn zwei Gruppen verschmolzen werden sollen. Genau hier liegt der entscheidende Vorteil zum vorher erwähnten Verfahren, zwei vorher unabhängige Listen können ohne vorheriges umständliches Kopieren verschmolzen werden. Vorsicht ist dennoch geboten, denn zwei Gruppen die erst einmal verschmolzen wurden lassen sich nur schwer wieder trennen.
Erweitert man Bobrows Technik der Gruppen um einen Pointer pro Zelle, über den eine lineare Liste aller Elemente der Gruppe geknüpft wird, so kann diese gesamte Liste in die Liste der freien Zellen eingehängt werden wenn die Gruppe freigegeben werden soll. Dieser Schachzug schließt den Bogen zum nicht- rekursiven Freigeben von Speicher. Beispielsweise könnte an dieser Stelle nun Weizenbaums *‚lazy freeing'* Algorithmus eingesetzt werden um eine Gleichverteilung der Last des Freigabezyklus zu erreichen.

```
/**
* Speichert p in das Feld f der Zelle
* c */
void store(pointer p, cell c){
   pointer temp = p;
   gp = groupNumber(p);
   gc = groupNumber(c);
   gt = groupNumber(temp);
   if(gp != gc){
      incrCount(p);
   }
   c.f = p;
   if(gt == gc){
      n = decrCount(temp);
      if(n == 0){
         reclaim(t)
      }
   }
}
```

Algorithmus 12: Speichern eines Pointers in eine Zelle

Einer der großen Nachteile am Bobrows Technik ist, dass man nur Zyklen behandeln kann die sich innerhalb einer Gruppe befinden, nicht aber solche die diese verlassen und wieder in sie zurückkehren.
Laut Hughes arbeiteten Bobrows Techniken am Besten wenn man den Graphen auf einzelne *‚strongly connected components'* herunter bricht. Das sind Teile des Graphen welche die Eigenschaft haben, dass jeder Einzelknoten von jedem anderen Knoten in der Komponente erreichbar ist. In diesem Fall kann jede Zelle sofort freigegeben werden sobald sie nicht mehr erreichbar ist [Hughes84]. Leider ist das aufteilen eines Graphen eine relativ aufwendige Arbeit, dennoch kann ein Graphen-reduzierungs- Algorithmus laut Hughes durchaus brauchbare Ergebnisse liefern.

4.3. Weak-pointer Algorithmen

Dieses Gebiet des Cyclic Reference Counting wurde bereits von vielen Autoren in Angriff genommen [Brownbridge85; Salkild87; Pepels88; Axford90]. Es beruht auf der Unterscheidung zwischen so genannten starken und schwachen (weak) Pointer. Als schwacher Pointer wird dabei jener bezeichnet der den Zyklus in einer zyklischen Datenstruktur schließt. Alle anderen

Pointer in solchen Strukturen werden als stark bezeichnet. Es resultiert also eine Struktur in der sich jeder Knoten im Heap von der Wurzel aus, über einen Pfad von starken Verbindungen erreichen lässt (*strongly reachable*). Das Wichtigste an dem Konzept ist sicherzustellen, dass es keine Kette von starken Pointern gibt die je einen Zyklus schließen können.

```
/**
* Erzeugt eine neue Zelle und gibt
* einen starken Pointer darauf zurück.
*/
pointer new(){
   /* kein freier Speicher mehr
    * vorhanden */
   if(freelist.empty()){
      abort("Memory exhausted");
   }
   newcell = allocateCell();
   /* strong reference count auf 1
    * setzen */
   SRC(newcell) = 1;
   //starker Pointer auf newcell
   return strongPointer(newnell);
}
```

Algorithmus 13: Zelle erzeugen nach Brownbridge

Zählt man nun nur die starken Referenzen so kann die herkömmliche Reference Counting Technik eingesetzt werden, da jeder aus starken Referenzen gebildete Graph zwingend azyklisch ist. Die Korrektheit von Weak-Pointer- Algorithmen hängt also von zwei Invarianten ab [Jones96]:

- Jeder aktive Knoten kann von der Wurzel her über einen Pfad von starken Referenzen erreicht werden.
- starke Referenzen dürfen niemals einen Zyklus schließen.

Am Weitesten verbreitet ist der Algorithmus nach Brownbridge. Dieser hat jedoch den Nachteil, dass er teilweise voreilig Objekte freigibt die eventuell noch benötigt werden. Brownbridge speichert in jeder Zelle zwei Referenzanzahlen. Eine speichert die starken Referenzen auf die Zelle und eine die Schwachen. Pointer zu neuen Zellen sind immer stark, weil sie ja keine Zyklen schließen können.
In Algorithmus 13 kann man erkennen wie im Algorithmus nach Brownbridge eine neue Zelle angelegt wird, auf die Frage wie Brownbridge's Pointersystem effizient realisiert werden kann wird später noch genauer eingegangen.
Aus Abbildung 8 wird ersichtlich in welchem Fall Brownbridge's Algorithmus beim Freigeben von Speicher voreilig reagiert. Die Unterstruktur die durch A, B und C gebildet wird, wird freigegeben sobald der starke Wurzelpointer nicht mehr auf den Knoten A verweist. Dadurch löst sich die gesamte vorher genannte Struktur von A her auf. Das hier auftretende Problem ist also jenes, dass ein Knoten nur mehr durch eine schwache Kante erreichbar ist. Verwendet man die oben angeführten Invarianten so ist unschwer zu erkennen, dass durch die Löschaktion des Pointers die Vorgabe verletzt wurde, dass jeder Knoten von der Wurzel aus über eine Kette von starken Zeigern erreichbar sein muss. Auf dieses Problem wird wenig später in diesem Artikel bei der Erklärung des verbesserten Löschalgorithmus noch einmal eingegangen.
Noch delikater wird die Angelegenheit wenn man das Kopieren von Pointern betrachtet. Beim Standardalgorithmus nach Brownbridge kann es passieren, dass durch den Kopiervorgang ein Zyklus geschlossen wird. Dabei ist es möglich dass alle Kanten dieses Zyklus starke Kanten sind, was laut der zweiten unserer Invarianten nicht passieren darf. Zumindest eine der Kanten (nach Brownbridge die Zyklus schließende) müsste zu einer schwachen Kante gemacht werden. Zu Bronwbridge's Ehrenrettung sollte an dieser Stelle jedoch erwähnt sein, dass sein Interesse nicht den allgemeinen Pointer- manipulations- Systemen galt sondern einen einem spezielleren Fall den ‚combinator machines'.
An dieser Stelle kommt Salkild ins Spiel der Brownbridges Algorithmus derart modifiziert, dass alle kopierten Pointer zu schwachen Pointern werden. Das hat natürlich direkt zur Folge, dass nun schwache Pointer überall im Graphen auftauchen können, nicht mehr nur dort wo sie Zyklen schließen. Auf den ersten Blick könnte man nun vermuten, dass dadurch eine der beiden Invarianten verletzt werden könnte. Das ist jedoch nicht der Fall wenn man die Erstellung einer neuen Zelle nach Brownbridges Algorithmus beibehält ist sichergestellt, dass zumindest ein starker Pointer auf jeden Knoten existiert. Ein Kopiervorgang kann auch die zweite Invariante nicht zu Fall bringen, da ja nur schwache Pointer produziert werden. Algorithmus 14 skizziert Salkilds Ansatz beim Update eines Pointers; Dabei drückt *WRC(C)* die Anzahl der schwachen Referenzen auf die Zelle *C* aus.

```
/**
* Umsetzen eines Pointers P auf die
* Zelle C */
void update(pointer P, cell C){
   delete(P);
   //weak reference count (WRC)
   WRC(C) = WRC(C) + 1;
   P = adress(S);
   //Pointer P schwach machen
   weaken(P);
}
```

Algorithmus 14: Update eines Pointers nach Salkild

Doch auch all diese Änderungen haben keinen Effekt auf das eigentliche Problem, das mit Abbildung 8 implizit angeführt wird. Das Löschen von Pointern wirft gewaltige Probleme auf. Die schwachen Pointer können in einem System das die vorher angeführten Invarianten erfüllt jederzeit gelöscht werden. Bei starken Pointern wird die Sache jedoch ungleich schwieriger. Wenn der letzte Pointer der gelöscht wurde ein starker war und gleichzeitig der Letzte (d.h. es existiert keine Referenz mehr auf die Zelle, weder stark noch schwach) so kann die Zelle ohne Bedenken freigegeben werden. Beim Freigeben einer Zelle sollte man darauf achten, dass auch alle Referenzen die die Zelle verlassen aufgelöst werden.
Der Dritte und letzte Fall beim Löschen einer Referenz ist, dass der gelöschte Pointer der letzte Starke auf eine Zelle ist, jedoch durchaus noch schwache Referenzen existieren. Das ist nun genau der Fall in dem der klassische Algorithmus teilweise versagt, denn an dieser Stelle würde die Zelle in jeden Fall freigegeben. Es ist jedoch möglich, dass der Knoten über eine Kette von starken und schwachen Pointern von der Wurzel aus erreichbar ist (was Invariante 1 nicht genügt!). Außerdem könnte noch eine starke Referenz zu einem anderen Knoten des Zyklus existieren, was bedeuten würde, dass alle Knoten des Zyklus noch erreichbar sind; Nur die schwache Kanten die nach Brownbridge eigentlich den Zyklus schließen sollte ist systematisch an der falschen Stelle. Algorithmus 15 skizziert den Löschalgorithmus der Alle der vorher erwähnten Fälle abdeckt.
Um zu ermitteln welcher der vorherigen Fälle zutrifft kann eine Suche von allen Nachfolgern der betreffenden Zelle durchgeführt werden um zu ermitteln ob eine von ihnen noch extern erreichbar ist; Dies ist natürlich gleichbedeutend mit der Gegebenheit, dass alle Knoten des Zyklus noch erreichbar sind.
Zuerst werden alle Pointer auf die Zelle deren Referenz gelöscht wurde, stark gemacht. Dadurch erreicht man, dass ermittelt werden kann ob die Zelle noch erreichbar ist oder nicht (natürlich zählen nun nur noch die starken Referenzen). Da dadurch eventuell Zyklen mit starken Pointern entstanden sind, müssen von der Zelle aus alle erreichbaren Knoten traversiert werden um einen solchen starken Zyklus gegebenenfalls auszumerzen. Dies wird durch die Unterroutine *suicide* erreicht. Nachdem diese eine Schlüsselposition in der Funktionsweise übernimmt soll sie in kürze näher erläutert werden. Algorithmus 16 zeigt deren Implementierung in der gewohnten Pseudocodedarstellung.
Die Funktion startet bei der Zelle deren Referenz entfernt wurde. Von dieser aus wurden wie in Algorithmus 15 gezeigt, bereits alle eingehenden Pointer gestärkt. Nun werden rekursiv alle ausgehenden starken Referenzen verfolgt.

```
/**
* Löschen eines Pointers P */
void delete(pointer P){
   //Fall 1: schwache Referenz
   if(isWeak(P)){
      WRC(P) = WRC(P)-1;
   }
   else{
      SRC(P) = SRC(P)-1;
      //Fall 2: keine Referenz mehr
      if(SRC(P)==0 && WRC(P)==0){
         forall(Children C of P){
            delete(C);
         }
         free(P)
      }
      //Fall 3: nur noch schwache Ref.
      else if(SRC(P)==0 && WRC(P)>0){
         invertStrength(P);
         forall(Children C of P){
            suicide(P,C);
         }
         if(SRC(P) == 0){
            forall(Children C of P){
               delete(C);
            }
            free(T);
         }
      }
   }
}
```

Algorithmus 15: Löschen von starken und schwachen Referenzen

Sollte es zu dem Fall kommen, dass durch diese Traversierung des Graphen der Ausgangsknoten wieder erreicht wird so hat man einen starken Zyklus gefunden. Um diesen zu eliminieren ist es nötig zumindest eine der Referenzen zu schwächen. Daher ist es nahe liegend den letzten, d.h. den schließenden Pointer zu schwächen, denn dieser ist noch direkt griffbereit. Doch genau dieser Ansatz birgt wiederum das Problem mit welchem Brownbridge von Beginn an zu kämpfen hatte.
Dadurch, dass genau der schließende Pointer geschwächt wird kommt es wieder zu einer voreiligen Freigabe. Siehe dazu noch einmal Abbildung 8a. Gehen wir davon aus, dass wiederum die Referenz von der Wurzel zum Knoten A gelöscht wird. Nun werden alle auf die Zelle A zeigenden schwachen Pointer zu Starken gemacht. Damit entsteht zwischen A, B und C ein starker Zyklus. Dieser wird natürlich über die Suchroutine *suicide* gefunden und dadurch beseitigt, dass die schließende Referenz (zwischen C und A) geschwächt wird. Dies ändert jedoch wiederum nichts an der Tatsache, dass die Struktur A, B und C nicht mehr über eine Kette von starken Zeigern

erreichbar ist, wie es von der ersten Invariante verlangt wird.

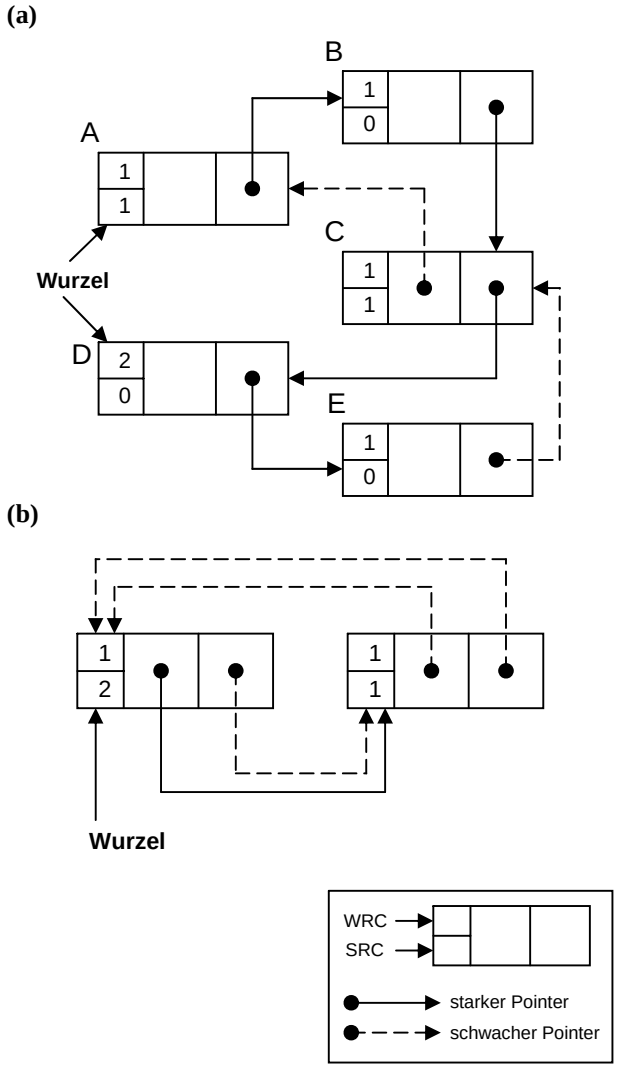


Abbildung 8: (a) Problemillustration Brownbridge (b) Problem bei Salkilds Verbesserung

Salkild lieferte eine Lösung für dieses Problem, indem er die *suicide* Routine modifizierte. Wenn eine Zelle erreicht wird die nur eine starke Referenz aufweist (über die sie erreicht wurde), allerdings jedoch andere schwache Referenzen hat, so sollte die Stärke aller ihrer Referenzen invertiert werden und die Suche nach externen Referenzen und starken Zyklen an dieser Stelle neu gestartet werden. Damit löste er Brownbridge's Problem der voreiligen Freigabe. Jedoch kaufte er sich dadurch eine neue Schwierigkeit ein, und zwar jene, dass seine Variante in manchen Fällen nicht terminiert. Ein solcher Fall ist relativ einfach generiert, indem man einfach zwei Zellen jeweils doppelt verknüpft und nur ein der Kanten (ausgehend von der extern erreichbaren Zelle) als stark deklariert. Abbildung 8b illustriert einen solchen Fall. Ein Ausweg für dieses Problem wird im Anschluss an den kleinen Einschub über die Realisierung skizziert.

Nun drängen sicht natürlich langsam die Fragen der Realisierung auf. Vor allem das Konzept der starken und schwachen Pointer läuft Gefahr das System unbrauchbar zu machen falls es nicht gewissenhaft, kosteneffektiv und laufzeiteffektiv implementiert wird.

Brownbridges Lösung zu diesem Problem sah wie folgt aus. Er versah jedes Objekt und jeden Pointer mit einem Stärke- Bit. Wenn Objekt und Pointer denselben Bitwert aufweisen, so ist die Referenz stark, ansonsten schwach. Diese Implementierung erscheint auf den ersten Blick überflüssig kompliziert hat jedoch einen besonderen Vorteil. So können beispielsweise alle schwachen Referenzen auf ein Objekt gestärkt und alle Starken geschwächt werden indem man einfach den Stärkewert des Objekts invertiert. Damit das gut funktioniert muss allerdings auch der Stärkewert bestimmen welche der beiden Referenzanzahlen die Starke und welche die Schwache ist. Realistisch einsetzten wird man den oben genannten Vorteil allerdings ohnehin eher, wenn nur noch schwache Referenzen existieren und diese zu starken gemacht werden sollen. Beispielsweise in Fall 3 von Algorithmus 15 werden alle schwachen Pointer zu Starken gemacht indem der Stärkewert des Objekts auf das P verweist invertiert wird, dies wird durch die schemenhafte Anweisung *invertStrength(P)* erreicht. Starke Referenzen existieren an dieser Stelle per Abfrage ohnehin nicht mehr.

```
/**
* Suchroutine die starke Zyklen von
* einem Knoten 'Start' aus findet und
* diese beseitigt. */
void suicide(pointer Start,
             pointer N){
   if(isStrong(N)){
      if(N == Start || SRC(N) > 1){
         weaken(N);
      }
      else{
         forall(Children T of N)){
            suicide(Start, T);
         }
      }
   }
}
```

Algorithmus 16: Suicide Suche die starke Zyklen unterbricht

Um das Thema der Weak- Pointer- Algorithmen gebührend abzuschließen soll an dieser Stelle noch ein Ausweg für das weiter oben genannte Dilemma des Nichtterminierens der *suicide* Suche skizziert werden.

Eine solche Lösung wurde von Pepels und seiner Gruppe vorgestellt [Pepels88].
Ihre Lösung beruhte grob umrissen darauf ein Markierungsschema zu verwenden. Es gibt dabei zwei Arten von Markierungen, jene die eine unendliche Anzahl von Suchen verhindern und Andere die garantieren, dass jede Suche terminiert. Leider erreicht ihr Algorithmus eine gewaltige Komplexität und auch Laufzeit- und Raumverhalten sind nicht optimal [Jones96]. Bei einem Szenario in dem es zu einer normalen Anwendung des Standard- Weak- Pointer- Kerns kommt, also keine starken Zyklen auftreten und jedes Löschen eines starken Pointers wirklich das Freigeben der Zelle bewirkt, ist der Algorithmus doppelt so langsam wie das typische Reference Counting Verfahren. Tatsächlich hat er jedoch sogar exponentielle Laufzeitkomplexität, auch wenn sie nur bei konstruiert wirkenden Fällen an den Tag tritt. Auch die Speicherkomplexität ist gegenüber anderen Reference Counting Verfahren nicht optimal. Immerhin benötigt man alles in Allen zwei Feldern in denen die Referenzanzahlen gespeichert werden, ein Stärkenbit für Objekt und Referenz sowie zwei Markierungsfelder. Der Vollständigkeit halber sollte noch erwähnt werden, dass laut Pepels diese beiden Markierungsfelder zusammengeführt werden können.

4.4. Lins partieller Mark-Sweep Algorithmus

Partielle Mark-Sweep Algorithmen unterscheiden sich grundsätzlich von den bisher erwähnten. Die Grundidee besteht darin die Datenstruktur in drei Phasen zu bearbeiten. Die erste Phase besteht darin den Einfluss von internen Referenzen ein einem Subgraphen zu eliminieren. Damit sind die Referenzanzahlen nun so modifiziert, dass sie nur noch Auskunft über die externen Referenzen auf Elemente des Graphen geben. In der zweiten Phase wird traversiert um festzustellen welche Knoten noch extern erreichbar sind, deren Referenzanzahlen werden wiederhergestellt. Die Dritte und letzte Phase beschäftigt sich nun damit, die nicht mehr benötigten Zellen in die Freelist zu verschieben. In Folge werden zwei wichtige Vertreter dieser Algorithmen näher beschrieben, wobei der Zweite (nächstes Kapitel) unter Umständen als ein Sonderfall des Ersten zu betrachten ist.
Der erste Algorithmus wurde von Lins und seiner Gruppe entwickelt [Jones96]. Dabei handelt es sich weitgehend um einen hybriden Algorithmus der den Großteil der Zellen über herkömmliches Reference Counting freigibt während Zyklen über einen Mark-Sweep Garbagecollector freigegeben werden. Nun stellt sich die Frage wie erkannt wird welcher Collector eingesetzt werden soll. Lins geht dabei so vor, dass eine Zelle auf die ohnehin nur eine Referenz existiert, über den Reference Counting Collector freigegeben wird sobald ihre Referenzanzahl auf null fällt. Wird allerdings eine Referenz auf eine mehrfach referenzierte Zelle gelöst, so untersucht der Mark-Sweep Collector die transitiven Enden des gelöschten Pointers. Martinezs Lösung zu diesem Thema ist aus Gründen der Laufzeitkomplexität (bei jedem entsprechenden Löschvorgang Traversierung des gesamten Subgraphen) nicht gangbar. Es wäre erstrebenswert eine Traversierung möglichst nur dann durchführen zu müssen wenn wirklich auch Zyklen freizugeben sind.
Lins entwickelte eine Möglichkeit diese Traversierungen aufzuschieben, dazu verwendet er sein so genanntes *Control Set*. Je nach Strategie kann diese Liste dann zu einem passenden Zeitpunkt durchsucht werden. Lins gibt dafür zwar eine spezielle Implementierung an, doch diese kann ohne weiteres durch eine zur Situation besser passende Implementierung ersetzt werden, doch dazu später.
Zuerst sollten wir Lins *lazy cyclic reference counting* Algorihmus etwas näher unter die Lupe nehmen. Wie erwähnt benutzt Lins sein Control Set um sozusagen potentiell freizugebende Zellen zu sammeln, und später abzuarbeiten. Der Unterschied zu normalen Mark-Sweep Algorithmen ist, dass dort nur aktive Datenstrukturen traversiert werden, während bei Lins Algorithmus im besten Fall nur zyklischer Müll durchsucht wird. Lins und seine Kollegen führen dazu zusätzlich zum Reference Count Feld noch ein weiteres Feld pro Zelle ein, das einen Zustand in Form von vier Farben speichert. Lins verwendet original grün, rot, blau und schwarz doch laut [Jones96] ist es eine gute Idee die Farben schwarz, grau, weiß und violett zu verwenden um mit dem im inkrementellen Garbagecollection verwendeten Schema kompatibel zu sein. Aktive Zellen sind dabei schwarz während freigegebene und freizugebende Zellen weiß markiert werden. Grau wird verwendet wenn eine Zelle markiert wird, also noch einmal abgearbeitet werden muss. Violette Zellen hingegen können beispielsweise Zellen eines Zyklus sein, diese müssen vom Garbage Collector selbst traversiert werden. Doch wie kommt man nun zu dieser Farbgebung?
Wenn eine Referenz zu einer mehrmals referenzierten Zelle gelöscht wird, so wird diese erstmal violett markiert und auch im Control Set vorgemerkt. Algorithmus 17 beschreibt wie die Löschfunktion in Lins System funktioniert. Violett verwendet man hierbei um zu vermeiden, dass Duplikate in das Control Set geschrieben werden, was sich jedoch nicht immer vermeiden lässt. Violett zeigt sozusagen gewissermaßen einen Unsicherheitsstatus an, wenn eine Zelle violett gefärbt und in das Set geschrieben wird, danach jedoch durch eine andere Aktion wieder umgefärbt wurde so verbleibt sie zwar im Set, ihr Status ist jedoch gewiss und muss nicht durch einen Collectoraufruf bestätigt werden. Lins geht bei seiner aufschiebenden Taktik davon aus, dass

sich der Zustand der Meisten im Control Set befindlichen Zellen ohnehin selbst definiert. Wenn beispielsweise die letzte Referenz auf eine Zelle gelöscht wurde so ist diese Zelle bereits freigegeben oder als freizugebend gekennzeichnet (also weiß). Oder eine Referenz auf eine Zelle wurde kopiert, wodurch sie zwangsläufig aktiv (also schwarz) ist. Wird jedoch zu der Zeit an der das Control Set abgearbeitet wird festgestellt, dass eine Zelle noch immer violett ist, so muss diese von einem lokalem Mark-Sweep Collector überprüft werden.

```
/**
* Löschen einer Referenz P auf eine
* Zelle */
void delete(pointer P){
   RC(P) = RC(P) -1;
   if(RC(P) == 0){
      colour(P) = black;
      forall(Children C of P){
         delete(C);
      }
      free(P);
   }
   else if(colour(T) != violet){
      if(ControlSet.full()){
         collectControlSet();
      }
      colour(P) = violet;
      ControlSet.addCell(P);
   }
}
```

Algorithmus 17: Löschen einer Referenz

Wenn eine Zelle neu erzeugt wird, so muss diese sogleich aktiv (schwarz) markiert werden. Das ist prinzipiell der einzige Unterschied zwischen Lins *new* und dem des standard Reference Counting.

```
/**
* Umsetzen eines Pointers P auf die
* Zelle C */
void update(pointer P, Cell C){
   delete(P);
   RC(S) = RC(S) + 1;
   P = address(S);
   //Entfernen von P,C vom Control Set
   colour(P) = black;
   colour(C) = black
}
```

Algorithmus 18: Umsetzen eines Pointers nach Lins

Algorithmus 18 beschreibt wie das Umsetzen eines Pointers in Lins System vollführt wird. Der einzige Unterschied ist jener, dass beide involvierten Zellen aktiv sein müssen und auch als solche markiert werden sollten. Wenn es die Implementierung zulässt sollten sie auch aus dem Control Set entfernt werden.
Um nicht mit dem System selbst den Speicher zu überfüllen ist es nötig Mehrfacheinträge in dem Control Set zu vermeiden. Die einfachste und eleganteste Lösung hierfür wäre das Verwenden einer Hashtable oder einer Bitmap. Was jedoch, wenn auf eine andere Struktur zurückgegriffen werden muss die eine solche Form von Ausschluss nicht unterstützt?

```
/**
* Traversieren eines Subgraphen von
der
* Zelle C aus. Modifiziert die
* Reference Counts am Weg so, dass sie
* den Subgraph nicht widerspiegeln und
* markiert alle diese Zellen grau. */
void markGrey(cell C){
   if(colour(C) != grey){
      colour(C) = grey;
      forall(Children T of C){
         RC(T) = RC(T) - 1;
         markGrey(T);
      }
   }
}
```

Algorithmus 19: Entfernt Referenzanzahlen innerhalb eines Subgraphen

Wie bereits erwähnt werden Duplikate in Lins Algorithmus dadurch vermieden, dass man die Zelle violett färbt. Eine violette Zelle wird nicht wieder ins Control Set geschrieben. Doch eine Zelle kann ihre Farbe durch andere Routinen wieder wandeln. Beispielsweise könnte die letzte Referenz auf die Zelle gelöscht worden sein, diese Zelle freigegeben und bereits wieder verwertet (schwarz) worden sein. Es könnte auch der Pointer kopiert worden sein, auch dadurch verfärbt sich die Zelle wieder schwarz. Kommt es nun zu einem neuerlichen Löschvorgang einer Referenz auf diese Zelle, dann kommt es zu Duplikaten im Control Set weil der Sicherheitsmechanismus über die Farbe Violett nicht mehr greift.
Wenn eine neue Zelle in das Control Set aufgenommen werden soll, dieses jedoch voll ist gibt es zwei Möglichkeiten, entweder man stößt den Mechanismus an der das Set abarbeitet, oder man erweitert die Menge. Ist die Menge beispielsweise als verkettete Liste am Heap implementiert, so kann das Control Set ohnehin nur voll werden wenn sich der Speicher des Heaps dem Ende zu neigt. In diesem Fall ist Garbage Collection ohnehin unausweichlich.
Eingangs wurde erwähnt, dass noch eine vierte Farbe existiert, die in dieser Erläuterung noch nicht genug Aufmerksamkeit bekommen hat. Diese Farbe ist Grau

und wird in der Markierungsphase verwendet um anzudeuten, dass die betreffende Zelle noch einmal besucht werden muss.

(a)
```
/**
* Durchgehen der grauen Zellen, um
* Garbage Zellen zu filtern und aktive
* wieder herzustellen. */
void scan(cell C){
   if(colour(C) == grey){
      if(RC(C) > 0){
         //Referenzen            wieder
herstellen
         scanBlack(C);
      }
      else{
         colour(C) = white;
         forall(Children T of C){
            scan(T);
         }
      }
   }
}
```

(b)
```
/**
* Wiederherstellen der Referenz-
* anzahlen die während markGrey
* verfälscht wurden und markieren der
* aktiven Zellen. */
void scanBlack(cell C){
   colour(C) = black;
   forall(Children T of C){
      RC(T) = RC(T) + 1;
      if(colour(T) != black){
         scanBlack(T);
      }
   }
}
```

Algorithmus 20: (a) filtern von aktiven und inaktiven Zellen (b) wiederherstellen der Referenzanzahlen

Die Methode *markGrey* die in Algorithmus 19 skizziert ist verfolgt einen Subgraphen und passt alle Referenzanzahlen derart an, dass sie die interne Verkettung im Subgraphen nicht mehr widerspiegeln. Damit wird ersichtlich welche dieser Knoten noch extern erreichbar ist; Diese sind dann jene Knoten die eine Referenzzahl größer als null haben. Um sicherzustellen, dass der Algorithmus terminiert werden die Zellen während dieses Schrittes grau gefärbt.
In einer zweiten Phase wird dieser graue Graph über die Methode *scan* noch einmal durchgegangen, diese wird in Algorithmus 20a skizziert. Die Funktion *scanBlack* (Algorithmus 20b) wird dabei dazu verwendet um die durch *markGrey* verfälschten Reference Counts wieder herzustellen. Zellen die keine externe Referenzen aufweisen, in dieser Phase also eine Referenzanzahl von null aufweisen, werden von *scan* weiß markiert um anzuzeigen, dass sie eventuell entsorgt werden müssen; Eine weiße Markierung ist jedoch nicht absolut, denn sie kann sich später wieder in ein Schwarze wandeln, wenn alle internen Referenzzahlen wieder hergestellt sind. Dies erfolgt beispielsweise durch die Funktion *scanBlack* welche ja die Referenzanzahlen wieder aktualisiert und auch die aktiven Zellen als solche markiert (schwarz).

```
/**
*   Freigeben   der   weiß   markierten
Zellen.
*/
void collectWhite(cell C){
   if(colour(C) == white){
      forall(Children T of C){
         collectWhite(T);
      }
      free(C);
   }
}
```

Algorithmus 21: Freigeben der weißen Zellen

In der dritten und letzten Phase tritt die Funktion collectWhite auf den Plan und sammelt alle Zellen die weiß verblieben sind ein und hängt diese in die Freelist. Algorithmus 21 gibt eine Beispielimplementierung zu dieser Methode an. Diese kann natürlich nach allen Regeln der Kunst verfeinert werden, an dieser Stelle des Algorithmus geht es nur mehr darum die bereits als Müll (weiß) erkannten Zellen freizugeben. Beispielsweise kann es in manchen Fällen sogar sinnvoll erscheinen den gesamten Heap frei zu räumen statt sich auf eine rekursive Traversierung einzulassen.

```
/**
* Hauptroutine des 3 Phasen
* Algorithmus nach Lins. */
void collectControlSet(){
   cell C = ControlSet.getNextCell();
   if(colour(C) == violet){
      markGrey(C);
      scan(C);
      collectWhite(C);
   }
   else if(!ControlSet.empty()){
      collectControlSet();
   }
}
```

Algorithmus 22: Drei Phasen Mark-Sweep nach Lins

Diese drei Phasen Vorgangsweise wie sie in Algorithmus 22 noch einmal programmtechnisch angegeben ist, entspricht wiederum der am Beginn dieses Kapitels angegebenen allgemeinen Vorgangsweise bei partiellen Mark-Sweep Algorithmen.

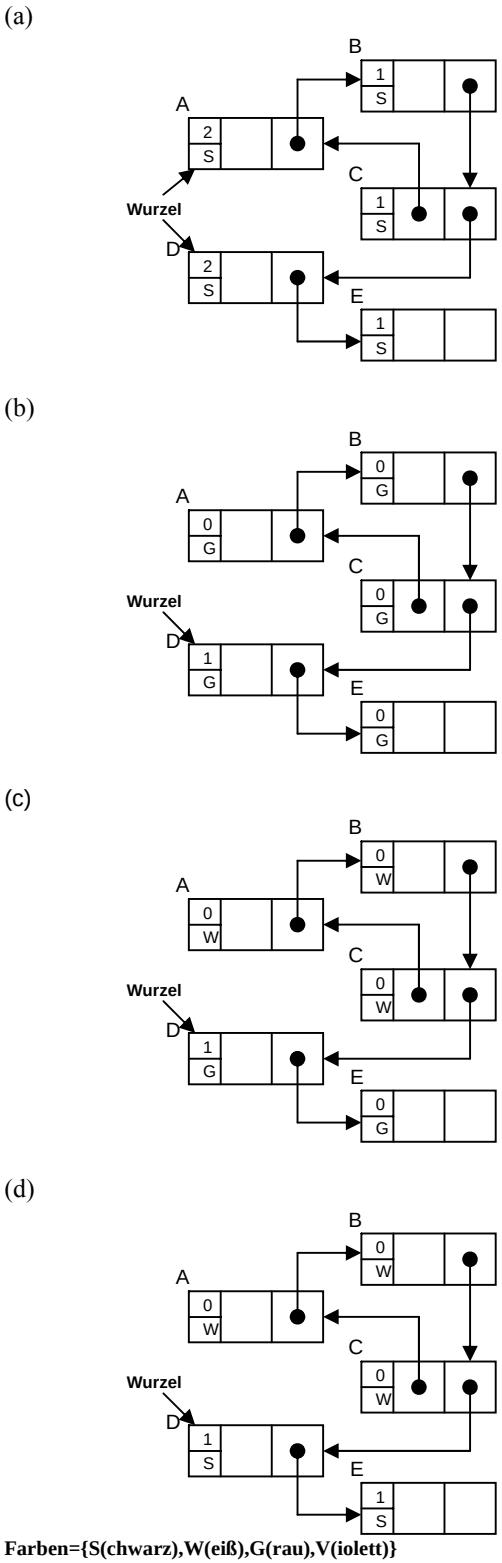
Die gesamte Vorgangsweise nach Lins führt ein Mark-Sweep nur dann aus, wenn eine neue Zelle benötigt wird, daher wird sie auch als „*Lazy Mark-Sweep*“ bezeichnet. Diese Eigenschaft hängt weitestgehend jedoch von der Strategie ab mit der das Control Set bewirtschaftet wird. Wie bereits zu Beginn erwähnt ist es möglich sich auch viele andere Strategien für eben diese Bewirtschaftung zu überlegen. Beispielsweise könnte man sich eine Strategie überlegen bei der der Collector immer nur dann angestoßen wird, wenn kein Platz mehr im Control Set ist, oder wenn keine freie Zelle mehr verfügbar ist, oder aber Beides. Außerdem könnte eine schrankenbasierte Lösung angestrebt werden, wenn die freien Zellen unter einen Grenzwert fallen soll der Collector aktiv werden. Die Strategie ist demnach nicht vorgegeben, sie kann also auch einer zum Anwendungsfall passenden Heuristik folgen.

Außerdem ist auch nicht vorgegeben wie viel des Control Sets auf einmal abgearbeitet wird. So ist es sicherlich für viele Anwendungen erstrebenswert eine gleich bleibende Last des Garbagecollectors zu erzeugen, weil man dadurch die Reaktionszeit des Systems besser abschätzen kann. Das erreicht man sicherlich am Besten dadurch, dass periodisch kleine Teile des Sets abgearbeitet werden. Andererseits kann es in einem anderen System ein anliegen sein so wenig Speicher im Heap wie möglich zu verbrauchen, was für einen möglichst frühen Sammelzyklus spricht.

Ebenso variabel ist aber auch die Realisierungstechnik des Control Sets selbst. So kann das Set als Stack, Queue, List oder auch Map realisiert werden. Im Extremfall kann eine solche Struktur den gesamten Zellenraum abdecken (Bitmap). Die Größe der Strukturen kann variabel oder fix sein.

Zusammenfassend kann man also über die vielen Variablen die Lins Control Set freilässt sicherlich eine Lösung zusammenstellen, die für den jeweiligen Anwendungsfall perfekt passt.

Laut [Jones96] stützt sich der Erfolg von Lins Methode vor allem auf die Annahmen, dass der Großteil der Zellen ohnehin nur einmal referenziert werden, also gar nicht in den Genuss des Mark-Sweep Teils kommen sondern sofort nachdem ihre letzte Referenz verloschen ist durch den Reference Counting Teil in die Freelist verschoben werden; Und außerdem noch auf die Eigenschaft, dass die Zyklen wenn sie auftreten hinreichend klein sind damit die Verzögerung durch das Traversieren nicht allzu groß wird.



Farben={S(chwarz),W(eiß),G(rau),V(iolett)}

Abbildung 9: Beispielablauf für Lins partiellen Mark-Sweep Algorithmus (a) Initialzustand (b) nach markGrey (c) vor scanBlack (d) vor collectWhite

Um dieses Kapitel über Lins Algorithmus gebührend abzuschließen wird in Abbildung 9 noch ein Beispiel für den Ablauf eines Freigabezyklus gegeben [Jones96].
Jede Zelle die in Abbildung 9 dargestellt wird besteht aus einem Feld für die Referenzanzahl, einem für die Farbe welche die Zelle gerade angenommen hat und zwei Zeigerfeldern die auf andere Zellen verweisen können.
Initial befindet sich alle Zellen im aktiven Zustand, sie sind alle erreichbar und damit schwarz gekennzeichnet (Abbildung 9a). Löscht man nun die Referenz von der Wurzel aus zum Knoten A, so wird dieser nachdem er mehrfach referenziert ist, violett gefärbt und in das Control Set aufgenommen.
Gehen wir nun davon aus, dass nach einer gewissen Zeit anderer Verarbeitung das Control Set abgearbeitet werden soll um eine neue Zelle freizugeben. Das Set liefert nun unsere noch immer violette Zelle A; Dabei ist angenommen, dass während der Verarbeitung keine Änderungen am Subgraphen passiert sind und auch A nicht modifiziert wurde. Wäre nämlich beispielsweise ein Zeiger auf A kopiert worden so hätte dies zum einen die Referenzanzahl, aber auch die Farbe der Zelle A beeinflusst. Wir haben nun also unsere violette Zelle A und nach Algorithmus 22 wird für diese erstmal ein *markGrey* fällig.
Dabei werden alle Zellen des Zyklus in der Reihenfolge ABCD und E traversiert und die Einflüsse der internen Referenzen auf die Referenzanzahl eliminiert. Gleichzeitig werden alle diese Zellen grau gefärbt, was zum einem das Terminieren des Algorithmus garantiert und zum Anderen das spätere Auffinden eben dieser Zellen erleichtert. Das Ergebnis dieses Schrittes ist ein Graph wie er in Abbildung 9b angegeben ist.
Die zweite Phase wird durch die Funktion *scan* eingeleitet, die nach Abschluss der *markGrey* Routine mit unserer Zelle A als Eingabewert aufgerufen wird. Diese Phase hat zum Ziel die nicht mehr benötigten Zellen von den noch aktiven Zellen zu trennen und außerdem die Referenzanzahlen wieder herzustellen. Abbildung 9c zeigt den Graphen nachdem die Knoten A, B und C traversiert wurden. Als nächstes kommt der Knoten D an die Reihe, der dazu führt dass die Unterroutine *scanBlack* aufgerufen wird. Diese markiert die Knoten D und E wieder als aktiv (schwarz) und stellt den internen Reference Count von E wieder her. Danach ist *scan* abgeschlossen, denn es existiert kein grauer Knoten mehr im Graphen. Abbildung 9d zeigt den Graphen nachdem die *scan* Funktion abgeschlossen ist.
Nun ist nur noch die letzte Phase abzuarbeiten in der die verbleibenden weißen Zellen A, B und C freigegeben werden.

4.5. Christophers partieller Mark-Sweep Algorithmus

Dieser Algorithmus kann als spezieller Fall von Lins Algorithmus betrachtet werden bei dem der gesamte Heap das Control Set bildet [Jones96]. Allerdings sind einige Designentscheidungen entscheidend anders getroffen worden was diesem Algorithmus einen anderen Reiz verleiht. Während Lins und seine Kollegen bei jeder ihrer Phasen auf das Traversieren eines Subgraphen von einer bestimmten Zelle aus setzen, geht Christophers Algorithmus mit einem linearen abarbeiten des gesamten Heaps an die Sache heran. Auch dieser Algorithmus arbeitet in drei Phasen die prinzipiell schrittweise genau denselben Zweck haben wie die bei Lins Algorithmus. Die Linearität der Phasen hat den entscheidenden Vorteil, dass man die Last besser abschätzen kann und der gesamte Algorithmus viel durchschaubarer läuft. Beim durchgehen eines Graphen hat man darüber hinaus das Problem, dass bei virtuellem Speicher die Performanz nicht so gut ist, weil man häufig von Seite zu Seite springt.
Darüber hinaus verwendet Christophers Algorithmus keinen zusätzlichen Speicher, man benötigt pro Zelle also nur das Reference Count Feld, das jedoch in der Markierungsphase für andere Zwecke genutzt wird. Der Algorithmus ist so entwickelt, dass er keine Unterstützung durch den Compiler benötigt.
Wie gesagt arbeitet der Algorithmus in drei Phasen. In der ersten Phase wird analog zur Routine markGrey aus Lins Algorithmus der Heap linear abgearbeitet und alle Referenzanzahlen reduziert die von Pointer herrühren deren Ausgangspunkt im Heap ist. Dadurch haben nur noch Zellen eine Referenzanzahl größer null die extern referenziert werden.
In der zweiten Phase werden alle Zellen die extern erreichbar sind als solche markiert (durch einen Eintrag in das Referenzanzahl Feld), inklusive deren Nachfolger.
Und in der dritten Phase werden die Referenzanzahlen wiederhergestellt und jene Zellen freigegeben die vorher nicht markiert wurden.

4.6. Zyklisches gewichtetes Reference Counting ohne Verzögerung

Die Idee dieses Algorithmus ist es, zwei Reference Couting Verfahren zu vereinen. Damit soll ein Algorithmus geschaffen werden der Garbage Collection in verteilten Anwendungen effizient möglich macht [Jones92].
Bei den beiden zu kombinierenden Verfahren handelt es sich einerseits um Jenes das bereits unter dem Namen *Lins partieller Mark-Sweep Algorithmus* besprochen wurde, und andererseits um das gewichtete Reference Counting welches ich hier in Grundzügen kurz umreißen werde. Ich werde jedoch auf genauere Ausführungen zu diesem Thema zugunsten einer längeren Beschreibung des eigentlichen Ziels dieses Kapitels verzichten. Zu Lins

Algorithmus ist noch zu sagen, dass es sich eigentlich um einen Vorgänger des von uns besprochenen Algorithmus handelt, dessen Änderungen aber so minimal sind, dass sie hier einfach vernachlässigt werden sollen.
Beim gewichteten Reference Counting wird zu jeder Referenz zusätzlich ein Gewichtswert mitgeführt. Ziel des Unterfangens ist es, ein System zu entwickeln, bei dem eine Änderung der Anzahl von Zeigern auf ein Objekt nicht zwangläufig in einer Änderung der Referenzanzahl resultiert; Das hat gerade bei verteilten Systemen den immensen Vorteil, dass der Kommunikationsaufwand gering gehalten werden kann falls die betreffenden Zellen sich beispielsweise in unterschiedlichen Prozessor- Heaps befindet.
Dazu wird im Referenzanzahlfeld der Zelle nicht die tatsächliche Anzahl der Referenzen mitgeführt, sondern die Summe der Gewichte dieser Referenzen.
Wie bereits in den vorangegangenen Algorithmen gesehen definiert sich ein Garbagecollection Verfahren vorwiegen über die Abarbeitung der Funktionen von *new*, *copy* und *delete*. Beim gewichteten Verfahren wird bei *new* einfach eine Zelle angelegt, deren Reference Count mit einer konstanten Zahl W belegt die eine Potenz von zwei ist, und die Referenz ebenfalls mit dieser Zahl W gewichtet. Nachdem davon ausgegangen werden kann, dass beide dieser Komponenten lokal erzeugt werden ist keine Kommunikation mit anderen Rechnerknoten notwendig.
Die Funktion *copy* die für uns noch nicht so geläufig ist kopiert einfach eine Referenz auf eine Zelle. Man geht dabei davon aus, dass die Zielzelle eventuell auf einem anderen Knoten im Netzwerk liegt. Um eine Kommunikation mit diesem Knoten zu vermeiden nutzt man nun die Eigenschaft der Gewichtung. Man halbiert das Gewicht der Ausgangsreferenz und gibt die verbleibende Hälfte an den neuen Knoten weiter. Dadurch ändert sich nichts am Reference Count Feld der Zelle und trotzdem wurde eine neue Referenz erzeugt. Zu einem Problem kommt es bei diesem System dann, wenn eine Referenz kopiert werden soll, dessen Gewicht bereits bis zur Unteilbarkeit auf eins gesunken ist. In diesem Fall wird eine Umleitungszelle erzeugt welche prinzipiell nur einen Zeiger mit Gewicht eins auf das ursprüngliche Referenzziel enthält. Die beiden Referenzen die eigentlich das ursprüngliche Ziel referenzieren sollten verweisen nun auf die Umleitungszelle, wobei mit der Gewichtsvergabe wieder von vorne begonnen wird; Jede der beiden Referenzen erhält damit das Gewicht W/2. Auch bei *copy* wurde dementsprechend keine eventuelle Kommunikation nötig, denn eine Änderung des Reference Count Feldes ist nicht nötig.
Die Funktion *delete* wird dadurch realisiert, dass die Referenz gelöscht wird und deren Gewicht von der Referenzanzahl der Zelle abgezogen wird. Befindet sich die Zelle also beispielsweise auf einem anderen Rechnerknoten so ist hier sehr wohl eine Kommunikation von Nöten. Sinkt die Referenzanzahl der Zelle auf *null* so kann davon ausgegangen werden, dass sie nicht mehr in Verwendung ist; Sie wird also in die Freelist verschoben.
Spätestens aus dieser Definition von *delete* wird das eigentliche Problem dieses Algorithmus deutlich. Er ist nicht in der Lage Zyklen aufzulösen.
Die Kombination dieses Algorithmus, mit dem vorher erwähnten partiellen Mark-Sweep Algorithmus nach Lins verspricht die Vorteile beider Verfahren zu kombinieren. So macht der gewichtete Teil eine geringe Kommunikation in verteilten Systemen notwendig, während der Mark-Sweep Teil auch Zyklen aus dem Speicher entfernen kann [Jones92].
Bei diesem Schema werden nur drei der vier Farben aus dem ursprünglichen partiellen Algorithmus verwendet. Nach Lins sind diese Farben *grün*, *rot* und *blau*, um jedoch zu unserer vorherigen Beschreibung kompatibel zu bleiben werden hier die Farben *schwarz*, *grau* und *weiß* verwenden. Die Farben bedeuten dabei wieder das Selbe wie in der vorherigen Beschreibung. Zusätzlich zu diesem Farbwert erhält die Zelle ein zweites Referenzanzahlfeld. Dieses wird nur vom Garbage Collector verwendet.
Wie gewohnt werden nun erst einmal die Basismethoden *new*, *copy* und *delete* vorgestellt. Dazu ist es allerdings nötig die bisher verwendete Pointer Notation etwas zu verfeinern. Dem Vorbild von [Jones92] folgend soll die Notation <S,T> für einen Zeiger von der Zelle S nach T stehen.
Wie in Algorithmus 23 skizziert, ist die Implementierung von *new* relativ klar aus der Gegebenheit ableitbar, dass es sich um eine Kombination der beiden Verfahren handelt.

```
/**
* Erzeugen einer neuen Zelle als Kind
* der Zelle C. */
void new(cell C) {
   U = new cell();
   RC(U) = W;
   Weight(<C,U>) = W;
   colour(U) = colour(C);
   SRC(U) = 0;
}
```

Algorithmus 23: Erzeugen einer neuen Zelle

Wie beim gewichten Verfahren wird die Referenzanzahl der Zelle mit einem konstanten Wert W belegt, der ebenso das Gewicht des Pointers zu dieser Zelle ist. Die neue Zelle übernimmt die Farbe jener Zelle, als deren Kind sie erzeugt wurde. Das wird gemacht damit die Zelle sich jeder Collector Phase anpasst die eben gerade

am laufen ist. Bei SRC(U) handelt es sich um den erwähnten zweiten Reference Count Wert der Zelle.

```
/**
* Kopiert den Zeiger <S,T> von R
* ausgehend sodass <R,T> entsteht */
void copy(cell R, pointer <S,T>){
   if(Weight(<S,T>) > 1){
      Weight(<S,T>) = Weight(<S,T>)/2;
      Weight(<R,T>) = Weight(<S,T>);
   }
   else{
      U = new cell();
      RC(U) = W;
      SRC(U) = 0;
      colour(U) = colour(S);
      Weight(<U,T>) = 1;
      Weight(<S,U>) = W / 2;
      Weight(<R,U>) = W / 2;
      controlSet.addCell(U);
   }
   if(colour(S)==grey&&MarkGreyPhase){
      send(retract(<S,T>),T);
   }
}

/**
* Korrigieren eines möglichen Fehlers
* in der MarkGreyPhase */
void handleRetract(pointer <S,T>){
   RC(T) = RC(T) + Weight(<S,T>)/2;
}
```

Algorithmus 24: Kopieren eines Zeigers

Auch das Kopieren einer Referenz ist sehr ähnlich der Lösung die man von einem System mit reinem gewichteten Reference Counting erwarten würde.
Im Fall, dass das Gewicht größer als 1 ist wird das Gewicht einfach aufgeteilt. Falls das Gewicht gleich 1 ist so muss eine Umleitungszelle eingefügt werden. Natürlich kann an diesem Verfahren noch einiges optimiert werden doch darum soll es in dieser Abhandlung nicht gehen. Algorithmus 24 zeigt wie das Kopieren von Referenzen in dem von Jones und Lins vorgestellten Algorithmus von statten geht.
Um sicherzugehen, dass es zu keinem Fehler in der markGrey Phase des Algorithmus kommt muss man falls der Knoten S bereits besucht wurde eine Berichtigung des Reference Counts von Knoten T veranlassen. Dazu sendet die letzte Funktion des Algorithmus eine Nachricht an den Knoten T. Eine Sendefunktion wird syntaktisch darum gewählt weil es sich um einen Knoten in einer anderen Netzkomponente handeln könnte.
Die letzte der drei charakteristischen Funktionen ist die *delete* Funktion. Diese wird genau abgearbeitet wie es die Kombination der beiden Verfahren vorgibt. Allerdings ist zu beachten, dass das Löschen eine zum Raum in dem die Zielzelle sich verbirgt, lokale Funktion ist. Daher wird das Löschen über eine einfache Nachricht ausgelöst und im jeweiligen Raum ausgeführt. Algorithmus 25 skizziert die beiden Routinen die demnach zum Löschen eines Zeigers nötig sind.

```
/**
* Sendet eine Nachricht an den lokalen
* Raum von S und löscht anschließend
* den Zeiger selbst. */
void delete(pointer <R,S>){
   send(deleteMessage(<R,S>), S);
   remove(<R,S>);
}

/**
* Handelt die Löschnachricht im
* lokalen Raum von S ab. */
void handleDelete(pointer <R,S>){
   RC(S) = RC(S) – Weight(<R,S>);
   if(RC(S) == 0){
      forall(Children T of S){
         send(deleteMessage(<S,T>),T);
      }
      colour(S) = black;
      free(S);
   }
   else{
      ControlSet.addCell(S);
   }
}
```

Algorithmus 25: Löschen einer Referenz

Analog zu Lins partiellen Mark-Sweep Algorithmus wird eine Zelle gelöscht falls ihr Reference Count Wert auf null fällt, anderenfalls wird die Zelle in das Control Set aufgenommen.
Auf Basis dieser drei Routinen können wir nun einen Schritt weiter gehen. Aus den partiellen Mark-Sweep Algorithmen ist bereits das dreiphasige vorgehen bekannt. Wie bei Lins ersten Algorithmus sind wir nun so weit, dass alle Kandidaten auf freizugebende Zellen in das Control Set geschrieben wurden. Ein kurzer Blick zurück auf Algorithmus 22 verrät, dass sich diese drei Phasen aus den Funktionen *markGrey*, *scan* und *collectWhite* zusammensetzen. Dieses Vorgehen findet auch hier seine Anwendung.
Die zusätzliche Herausforderung die sich nun jedoch durch das zugrunde liegende verteilte System ergibt ist folgende. Es kann sein, dass mehrere der Elemente des Systems gleichzeitig Garbage Collection durchführen wollen. In diesem Fall muss sichergestellt werden, dass sich diese untereinander nicht in die Quere kommen.

```
/**
* Traversieren des Subgraphen unter S
* sowie dessen markieren und
* modifizieren des SRC */
void markGrey(cell S){
   if(colour(S) != grey){
      colour(S) = grey;
      SRC(S) = RC(S);
      forall(Children T of S){
         send(msgMarkGrey(<S,T>),T);
      }
   }
}

/** Behandeln der MarkGrey Nachricht
*/
void handleMarkGrey(pointer <S,T>){
   if(colour(T) != grey){
      colour(T) = grey;
      SRC(T)      =      RC(T)      -
Weight(<S,T>)/2;
      forall(Children U of T){
         send(msgMarkGrey(<T,U>),U);
      }
   }
   else{
      SRC(T) = SRC(T) – Weight(<S,T>);
   }
}
```

Algorithmus 26: Markierungsphase, verschleiern interner Verbindungen

Für eine solche Aufgabe liegt es natürlich auf der Hand, dass es multiple Lösungen gibt. Eine der einfachsten Lösungen für dieses Problem ist es eine Synchronisation einzuführen. Darf beispielsweise ein Prozessor erst mit der *markGrey* Phase beginnen wenn der derzeit ausführende Prozessor die *collectWhite* Phase abgeschlossen hat, so kann es zu keinen Problemen wegen falsch markierter Zellen kommen. Dieses Muster wäre beispielsweise über Token passing Verfahren leicht zu implementieren.
Eine andere Möglichkeit der Synchronisation ist es ein Rendezvous System zu benutzen. Will beispielsweise eine Untergruppe der gesamten Rechnerknoten eine Collection durchführen so wird einfach gewährleistet, dass keiner der Knoten eine neue Phase beginnt bevor alle Anderen die aktuelle Phase abgeschlossen haben. Wird das für alle drei Phasen durchgezogen kommt es zu keinen Fehlern; Dazu müssen allerdings noch einige anderen Synchronisationsmechanismen eingehalten werden, beispielsweise darf kein Prozessor eine Collection nachträglich starten, bevor die andere Untergruppe ihren Vorgang abgeschlossen hat.

(a)
```
/**
* Filtern von verwendeten und nicht
* mehr erreichbaren Zellen. */
void scan(cell C){
   if(colour(C) == grey){
      if(RC(C) > 0){
         scanBlack(C);
      }
      else{
         colour(C) = white;
         forall(Children T of C){
            send(msgScan(T),T);
         }
      }
   }
}

/** Behandeln der Scan Nachricht */
void handleScan(cell T){
   scan(T);
}
```

(b)
```
/**
* Rückfärben des erreichbaren Graphen
*/
void scanBlack(cell C){
   if(colour(C)!= black){
      colour(C) = black;
      forall(Children T of C){
         send(msgScanBlack(T),T)
      }
   }
}

/**
* Behandeln der ScanBlack Nachricht */
void handleScanBlack(cell T){
   scanBlack(T);
}
```

Algorithmus 27: (a) filtern aktiver und nicht erreichbarer Zellen (b) Rückfärben aktiver Zellen

Die Funktion *markGrey* unterscheidet sich von jener die bereits in vorangegangen Kapiteln beschrieben wurde nur wenig. Trotzdem wird die Vorgangsweise in Algorithmus 26 kurz skizziert. Die hauptsächliche Änderung betrifft das Fakt, dass man nicht sicher sein kann ob ein Knoten in lokalen Speicher liegt oder anderswo, deshalb werden Nachrichten und Nachrichtenbehandlungsroutinen verwendet um diesen Dilemma zu entgehen.
Eingangs wurde erwähnt, dass jede Zelle ein zweites Reference Count Feld aufweist. Dieses wird dazu verwendet um den eigentlichen Reference Count während der Phasen nicht verändern zu müssen. Alle Operationen

die sonst direkt am Referenzanzahlfeld durchgeführt wurden werden nun im zweiten Feld durchgeführt, was zur Folge hat, dass ersteres nicht mehr wiederhergestellt werden muss.

```
/**
*   Freigeben   der   weiß   markierten
Zellen.
*/
void collectWhite(cell C){
   if(colour(C) == white){
      forall(Children T of C){

send(msgCollectWhite(<C,T>),T);
         remove(<C,T>);
      }
      free(C);
   }
}

/**
* Behandeln der CollectWhite Nachricht
*/
void         handleCollectWhite(pointer
<C,T>){
   if(colour(T)== white){
      collectWhite(T);
   }
   else{
      handleDelete(<C,T>);
   }
}
```

Algorithmus 28: Freigeben der weiß markierten Zellen

Die Funktion *scan* hat sich gegenüber unserer Referenzimplementierung beinahe überhaupt nicht verändert; Der einzige Unterschied besteht darin, dass die rekursiven Aufrufe aufgrund der verteilten Struktur unseres Systems wieder über das bereits bekannte Nachrichtensystem weitergegeben werden. Der Vollständigkeit halber soll sie jedoch zusammen mit der Funktion *scanBlack* in Algorithmus 27 noch einmal angeführt werden. Letztere Funktion hat in ihrer Kompetenz eingebüßt nachdem das zweite Reference Count Feld eingeführt wurde, ist das wiederherstellen des ursprünglichen Reference Count Wert hinfällig geworden. Die Funktion färbt demnach nur mehr die erreichbaren Zellen wieder schwarz (aktiv).
Um den drei Phasen Zyklus abzuschließen fehlt nun nur noch die *collectWhite* Phase. Nach den Erklärungen zum Vorangegangen sollte deren Funktion klar sein. Algorithmus 28 skizziert diese im gewohnten Pseudocode.
Im Kapitel über Lins partiellen Mark-Sweep Algorithmus wurde die Variabilität des Control Sets besprochen. Dieselben Möglichkeiten sind auch hier gegeben. Kombiniert mit den verschieden Implementierungsvarianten der Synchronisation ist das gesamte System noch variable als erstere Variante. Beispielsweise wäre es auch möglich die Garbage Collection von einem einzigen Rechnerknoten aus zu machen, der nur diesen einen Task ausführt, damit wäre annähernd eine realtime Performance möglich [Jones92].
Allerdings sollte an dieser Stelle noch einmal erwähnt werden, dass einige Rahmenbedingungen für die Synchronisation und die Kommunikation zwischen den Recheneinheiten gegeben sein müssen. Bei normalen gewichteten Reference Counting kommt es ausschließlich bei dem Löschen zu einer Kommunikation zwischen verschiedenen Knoten. Der große Vorteil dieses Schemas ist, dass keine Synchronisation zwischen den Knoten nötig ist. Beim hier präsentierten erweiterten Algorithmus ist es jedoch wie bereits erwähnt nötig, dass die einzelnen Phasen untereinander synchronisiert sind. Eine Lösung für dieses Problem würde eine zentrale Kontrolleinheit bieten. Ist das ob der Systemstruktur nicht möglich kann auch auf multicast Protokolle zurückgegriffen werden.

5. Resümee

Am Ende stellt sich immer die Frage ob es sinnvoll ist auf ein System wie Reference Counting zurückzugreifen, oder ob man doch eine andere Art des Garbage Collection vorziehen sollte. Im Laufe dieser Arbeit wurden die hauptsächlichen Probleme des Verfahrens angeschnitten und Lösungsversuche dazu präsentiert. So kann beispielsweise die Verzögerung die durch das rekursive Freigeben von Strukturen entsteht durch eine „faule“ Abarbeitung eben jener gemildert werden. Oder das Problem mit den zyklischen Datenstrukturen durch immer gewitztere Verfahren gelöst werden. Aber beispielsweise der Speicheroverhead der durch das Reference Count Feld entsteht wird nie vollständig aufgelöst werden können. Außerdem ist die Ausführungszeit die von referenzzählenden Verfahren verbraucht wird generell höher als jene von verfolgenden Verfahren. [Jones96]
Auf die andere Seite der Waagschale stellen sich jedoch Andere nicht weniger gewichtige Gründe. Beispielsweise erscheint die Implementierung von Referenzzählenden Verfahren meist als einfacher. So müssen generell nur Pointer Zuweisungen so manipuliert werden, dass sie die für Reference Counting nötigen Nebenaktionen mit ausführen. Ein weiteres Pro für diese Technik ist, dass man es mit programmiertechnischer Freigabe von Zellen vermischen kann. Dabei ist jedoch große Vorsicht geboten. Denn obwohl sich referenzzählende Verfahren leicht implementieren lassen ist es wichtig, dass sie in Tritt bleiben. Wird beispielsweise der Reference Count einmal fälschlicherweise erhöht, so ist es relativ gewiss,

dass die Zelle im gesamten Lebenszyklus der Garbage Collectors nicht mehr freigegeben wird.
Ein weiterer Punkt der sehr für die Verwendung von referenzzählenden Techniken spricht ist, dass sich ihr Rechenaufwand mit dem den Nutzprogramms beinahe perfekt vermischt. Werden zusätzlich „faule“ Freigabeverfahren verwendet verzahnt sich der gesamte Reference Counting Aufwand mit anderen Programmen.
Das sicherlich größte Problem bei Referenzzählenden Verfahren ist jedoch jenes der zyklischen Datenstrukturen. Algorithmen um diese dennoch aus dem Speicher zu entfernen wurden in den letzten Kapiteln dieser Arbeit besprochen. Dabei sollte jedoch erwähnt werden, dass viele dieser Möglichkeiten an Unvollständigkeit kränkeln, oder an der schlichten Tatsache, dass sie nicht terminieren. Das Fehlen von Lösungen für dieses Problem wird nicht zuletzt dadurch ausgedrückt, dass häufig Mark-Sweep Collectoren zusätzlich verwendet werden um diese Schwäche auszugleichen. Außerdem ist es nötig zu erwähnen, dass keine der präsentierten Algorithmen je in einem größeren kommerziellen Projekt eingesetzt wurde [Jones96].
Dennoch gibt es immer wieder Projekt die Reference Counting als nutzbringende Garbage Collection Technik zu etablieren versuchen. So wurde beispielsweise erst im Jahr 2001 eine Garbage Collector Implementierung Namens *Recycler* von Bacon und seinen Kollegen entwickelt [Bacon01].
Dabei handelt es sich um einen rein referenzzählenden Garbage Collector, der in der Lage ist Zyklen zu erkennen. Dazu wird ein Markierungssystem über Zustände beziehungsweise Farben ähnlich wie bei Lins Algorithmus verwendet. Dieser Collector wurde in der Jalapeńo Java virtuelle Maschine implementiert die auf Multiprozessoren mit gemeinsamem Speicher läuft.
Dieser erneute Versuch schnitt bei Benchmarks im Vergleich zu Mark-Sweep Collectoren nicht schlecht ab. Bei einem einzelnen Prozessor lief der Recycler immerhin mit ca. 90% der Geschwindigkeit des Mark-Sweep Collectors. Seine wirklichen stärken entfaltete er jedoch erst beim hinzufügen eines weiteren Prozessors der das Garbage Collection übernahm sowie bei signifikanter Erhöhung des Speichers.

6. Literatur:

[Collins60] Collins, George E.; A method for overlapping and erasure of lists. CACM 3(12):655-657, 1960

[Gelernter60] Gelernter, H.; Hansen, J. R.; and Gerberich, C.L.; A Fortran-compiled list processing language; J. Assoc. Comput. Mach. 7; 1960

[McCarthy60] Mc Carthy, J.; Recursive functions of symbolic expressions and their computation by machine, Part I. Commun. ACM 3; 9160

[McBeth63] McBeth, H.: On the reference counter method. Communications of the ACM, 6(9):575, 1963

[Weizenbaum63] Weizenbaum, Joseph; Symmetric List Processor. Comm. ACM 6, 9 (Dec. 1963)

[Weizenbaum69] Weizenbaum, Joseph; Recovery of reentrant list structures in SLIP; Communications of the ACM, v.12 n.7; July 1969

[Knuth73] D.E. Knuth and F.R. Stevenson. Optimal measurement points for program frequency counts. BIT, vol. 13, 1973

[DeutschBobrow76] Deutsch, L. Peter; Bobrow, Daniel G.; An efficient incremental automatic garbage collector; Communications of the ACM, 19(9):522--526, September 1976

[Clark77] Douglas W. Clark and C. Cordell Green, An Empirical Study of List Structure in Lisp, Communications of ACM, Vol. 20, No. 2., February 1977

[FriedmanWise77] Wise, David S.; Friedman, Daniel P.; The one-bit reference count. BIT, 17(3):351--9, 1977. 90

[Friedman79] Friedman, D., Wise, D.: Reference counting can manage the circular environments of mutual recursion, Information Processing Letters 8 41-44, 1979

[Wise79] Wise, Daniel S.; Reference counting can manage the circular environments of mutual recursion; Jan 1979

[Bobrow80] Bobrow, D.: Managing Reentrant Structures Using Reference Counts. ACM TOPLAS 2(3):269-273, 1980

[Baden83] Baden, Scott B.; Low-overhead storage reclamation in the Smalltalk-80 virtual machine; 1983

[Stoye84] Stoye, W.R., Some practical methods for rapid combinator reduction. *Lisp & Funct. Progr. Conf. 1984.*

[Hughes84] Hughes, J.: Reference Counting with circular Structures in virtual Memory, applicative Systems. Technical Report. Programming Research Group, Oxford University, 1984

[Brownbridge85] Brownbridge: Cyclic Reference Counting for Combinator Machines, in Functional

Programming Languages and Computer Architectures. Lecture Notes in Computer Science 201, Springer Verlag 273-288. 1985

[Wise85] Wise, David S.; Design for a multiprocessing heap with on-board reference counting;. Springer-Verlag. In J.-P. Jouannaud (ed.), Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201: 289—304; 1985

[Taylor86] Taylor, George S.; Hilfinger, Paul N.; Larus, James R.; Patterson, David A.; Zorn, Benjamin G.; Evaluation of the SPUR Lisp Architecture, Proc. ISCA '86, pp. 444-452, 1986

[ChikayamaKimura87] T. Chikayama; Y. Kimura; Multiple reference management in Flat GHC. ICLP, pages 276-293, 1987

[Salkild87] Salkild, J.: Implementation and analysis of two reference couting algorithms. Master' thesis, University College, London, 1987

[Hartel88] Pieter H. Hartel. Performance Analysis of Storage Management in Combinator Graph Reduction. PhD thesis, Department of Computer Systems, University of Amsterdam, Amsterdam, 1988.

[Pepels88] Pepels, E, Eekelen, M, Plasmeijer, M.: A cyclic Reference Counting Algorithm and ist Proof. Technical Report 88-100, Computing Science Department, Univeristy of Nijmegen, 1988

[Appel89] Appel, A. W.; Simple Generational Garbage Collection and Fast Allocation; 1989

[Zorn89] Comparative Performance Evaluation of Garbage Collection Algorithms. Computer Science Division (EECS) of University of California at Berkeley. Technical Report UCB/CSD 89/544

[Axford90] Axford, T.: Reference Counting of Cyclic Graphs for Functional Programs. The Computer Journal 1990 33(5):466-470. 1990

[Hayes91] Using key object opportunism to collect old objects. In Proceedings of OOPSLA'91 Conference on Object-Oriented Systems, Languages and Applications, ACM SIGPLAN Notices 26(11), Phoenix, Arizona. ACM Press, pages 33-46. October 1991

[Appel92] Smartest Recompilation. Zhong Shao and Andrew W. Appel. CS-TR-395-92, Princeton University, 1992. Proc. Twenthieth ACM Symp. on Principles of Programming Languages January 1993.

[PeytonJones92] Peyton Jones, Simon L.; The Glasgow Haskell compiler: a technical overview. 1992

[Jones92] Jones, R, Lins, R.: Cyclic weighted reference counting without delay. Computing Laboratory, The University of Kent at Canterbury. Technical Report 28-92. 1992.

[BarretZorn93b] The Measured Cost of Conservative Garbage Collection; Benjamin Zorn, David Barrett; Department of Computer Science, Campus Box 430, University of Colorado, USA, 1993

[GehringerChang93] Gehringer, Edward F.; Chang, Ellis; Hardware-assisted memory management; 1993

[Wise93] Wise, David S.; Stop-and-copy and one-bit reference counting. Information Processing Letters. 46, 5; 1993

[Baker94] Baker, Henry G.; Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures; 1994

[Wise94] Wise, David S.; Heck, Brian; Hess, Caleb; Hunt, Willie; Ost, Eric; Uniprocessor performance of a reference-counting hardware heap. Technical Report TR-401, Indiana University, Computer Science Department, May 1994

[Jones96] Jones, R, Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley 1996

[Bacon01] Bacon, D., Attanasio, C., Loc, H, and Smith, S. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. PLDI'01, SIGPLAN Notices, 2001