Automatisches Speichermanagement

Automatisches Speichermanagement

Viele moderne Programmiersprachen verfügen über automatisches Speichermanagement.

Durch höhere Sprachkonstrukte, wie zum Beispiel *Closures*, wird manuelles Speichermanagement zunehmend komplizierter. So gut wie alle funktionalen Programmiersprachen haben automatisches Speichermanagement.

Der größte Teil der neu entwickelten Programmiersprachen ist auf automatische Speicherverwaltung eingerichtet.

$\Rightarrow$ Automatische Speicherverwaltung ist ein wichtiger Aspekt der Compilertechnologie.

Garbage Collection

Automatische Speicherverwaltung durch Garbage Collection

1. Identifizierung aller *lebendigen* Objekte im Speicher
Ein Objekt ist lebendig, wenn in der Zukunft noch darauf zugegriffen wird. Die Lebendigkeit von Objekten ist unentscheidbar und kann somit nur approximiert werden. D.h. der Garbage Collector wird i.a. auch einige Objekte als lebendig ansehen, die schon gelöscht werden könnten.

2. Rückgewinnung von Speicherplatz
Speicherplatz, der von nichtlebendigen Objekten („Müll“) belegt wird, muss dem Programm wieder zur Verfügung gestellt werden.

Lebendigkeit — Beispiel

```
List y = new List(1,x);
{
 List x = new List(5,3);
} // x ist jetzt nicht mehr lebendig

{
 List x = null;
 for(int i;i <= 10;i++) {
    x = new List(i,x);
 }
} // x etc ist nicht mehr lebendig

{
 List x = new List(5,new List(4,new List(3,null)));
 y = x.tl;
} // x ist nicht lebendig, x.tl aber schon.
```

Garbage Collection

Möglichkeiten zu Implementierung von Garbage Collection:

- *Reference Counting*
- *Mark-and-Sweep*
- *Copying Collection*
- *Generational Garbage Collection*
- *Incremental Garbage Collection*

Referenzzählen

Jedes Objekt erhält einen Zähler (reference counter), in dem die Zahl der auf das Objekt verweisenden Referenzen gespeichert wird. Wird dieser Zähler 0, so kann das Objekt freigegeben werden.

- Jedes Objekt erhält ein zusätzliches Feld `count`.
- Bei Allokierung mit `new` wird `count` auf 1 gesetzt.
- Bei Zuweisung `x.f = y` setze `y.count++; x.f.count--;`.
- Wird ein Referenzzähler 0, so füge das Objekt der Freelist hinzu und dekrementiere die Zähler aller Felder.

Referenzzählen

Nachteile

- ineffizient
- zyklische, aber von außen nicht erreichbare, Datenstrukturen schwer zu entdecken

Vorteile

- inkrementelle Speicherfreigabe während der Programmausführung, wichtig z.B. für Echtzeitanwendungen
- Anwendbarkeit in verteilten Systemen (z.B. Java RMI)

Für die Implementierung von Programmiersprachen wie Java wird das Referenzzählen wegen seiner Ineffizienz nur selten eingesetzt.

Mark and Sweep

Garbage Collection mit dem Mark-and-Sweep Verfahren.

Allokiere neue Objekte so lange bis der Speicher voll ist.

Ist der Speicher voll, werden nichtlebendige Objekte mit folgendem Verfahren aus dem Speicher entfernt:

- Markiere (**mark**) mittels Tiefensuche alle Objekte, die von den in Programmvariablen gespeicherten Werten (Wurzelobjekte) erreichbar sind.
  Beachte: Es werden alle lebendigen Objekte markiert, eventuell aber auch nichtlebendige.
- Entfernung aller nichtmarkierten Objekte (**sweep**).

Mark and Sweep

Jedes Objekt erhält ein zusätzliches Boolesches Feld `mark`.
Rekursive Implementierung:

```
DFS(x) {
  if (x != null && !x.marked) {
    x.marked = true;
    for (f in Felder(x)) {DFS(x.f);}
  }
}

SWEEP(){
  p = erste Adresse im Heap;
  while (p < letze Adresse im Heap) {
     if (p.marked) { p.marked = false; }
     else { Fuege p in Freelist ein; }
     p = p + p.size;
  }
}
```

Optimierung von Mark and Sweep

Die Rekursionstiefe von `DFS` kann sehr groß werden, z.B. bei sehr langen verketteten Listen.

Benutzung eines expliziten Stacks statt Rekursion reduziert den Speicherbedarf, da statt eines ganzen Stack-Frames nur noch eine Adresse pro Stackeinheit gespeichert wird.

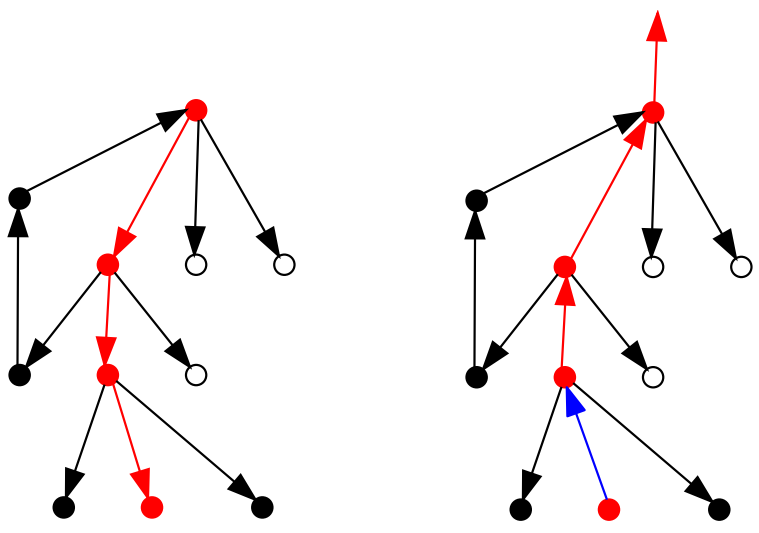
Dieser Stack kann immer noch recht groß werden und muss auch irgendwo gespeichert werden (GC wird typischerweise durchgeführt, wenn der Speicher voll ist).

Der Stack kann im Objektgraph selbst kodiert werden:
Pointer-Reversal [Deutsch, Schorr, Waite]

Pointer Reversal

Bei der Tiefensuche wird jede Kante einmal vorwärts und dann noch einmal rückwärts durchlaufen. Da wir Pointern nur in eine Richtung folgen können, wird das Rückwärtslaufen durch den Stack realisiert.

Optimierungsidee: Drehe die Kante beim Vorwärtsdurchlauf um, so dass sie auch ohne Stack rückwärts durchlaufen werden kann, und stelle sie dann wieder her.

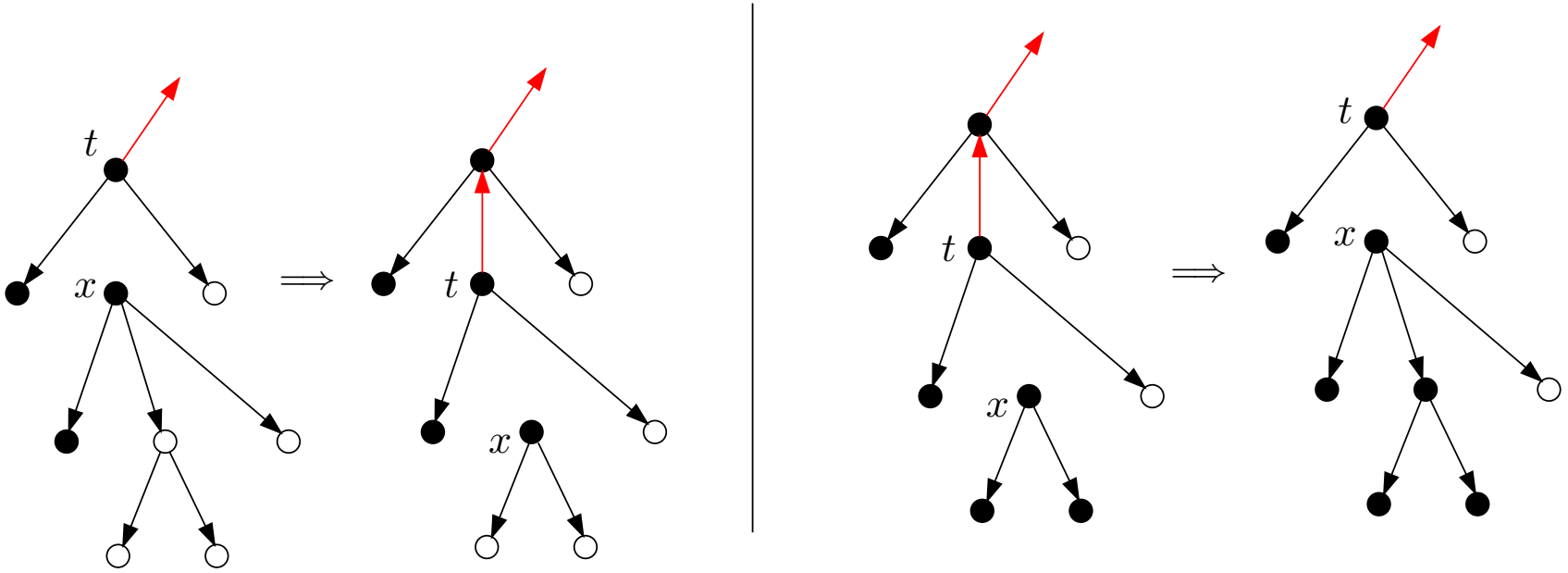


Pointer Reversal

In jedem Knoten wird gespeichert:

- ein Bit zur Markierung schon besuchter Knoten (im Bild gefüllt)
- eine Zahl, die angibt, welcher der ausgehendenen Zeiger des Knotens umgedreht wurde (im Bild rot)

Zum Durchlaufen hält man dann einen Zeiger x auf das aktuelle Objekt und einen Zeiger t auf das vorangegangene Objekt.



Mark and Sweep

Vorteile

- effizienter als Referenzzählen
- Objekte werden nicht verschoben

Nachteile

- Speicherfragmentierung
  (kann durch Mark and Compact verbessert werden)
- In der Sweep-Phase wird der gesamte Speicher durchlaufen.
  Die Kosten eines GC-Laufs sind proportional zur
  Speichergröße, nicht nur zum benutzten Speicher.

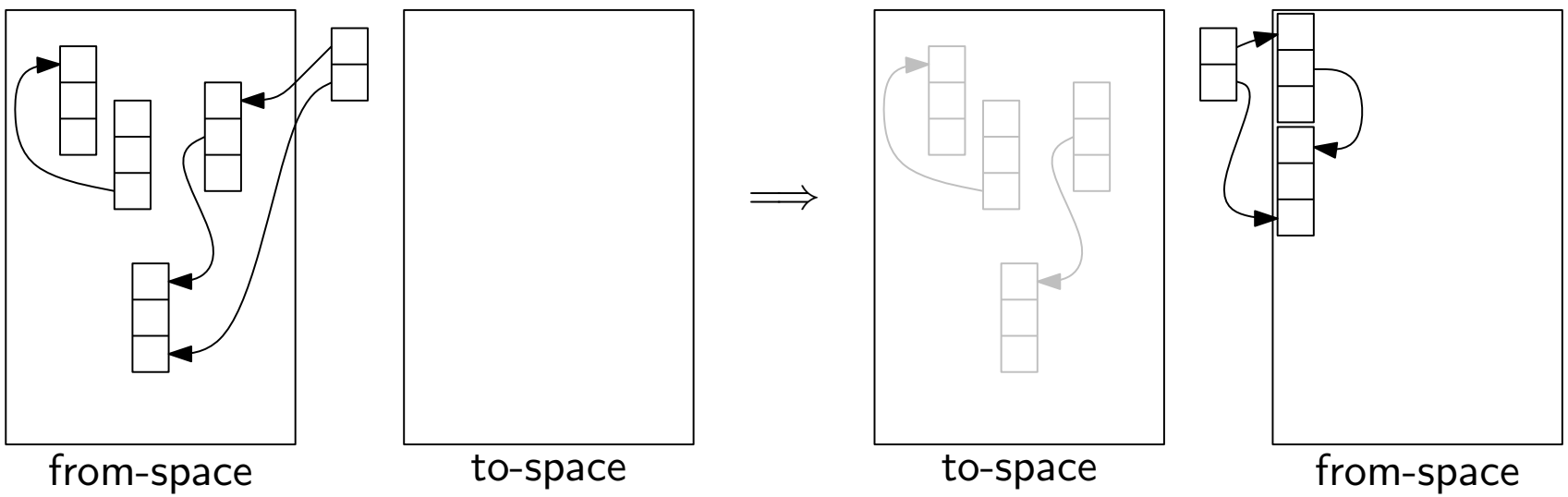
Copying Collection

Aufteilung des Speichers in zwei gleich große Bereiche (*from-space*, *to-space*).

Allokierung neuer Objekte im *from-space*.

Wenn der Speicher knapp wird:

- Kopieren aller erreichbaren Objekte vom *from-space* nach *to-space*.
- Vertauschen von *from-space* und *to-space*.



Copying Collection

Vorteile

- Aufwand für GC hängt nur von der Menge der erreichbaren Objekte ab, nicht von der Gesamtgröße des Speichers.
- keine Freelist nötig
- keine Fragmentierung

Nachteile

- doppelter Platzbedarf

Copying Collection — Cheney Algorithmus

Der *to-space* erhält zwei Zeiger: *next* (nächste freie Adresse), *scan* (Objekte unterhalb dieser Adresse wurden komplett verarbeitet, d.h. alle von dort erreichbaren Objekte sind schon im *to-space*).

Ablauf der Kopieroperation:

- Zu Beginn: Kopiere alle Wurzelobjekte (erreichbar von Variablen) in den *to-space*; schreibe Adresse des neuen Objekts als *Forward-Pointer* in das erste Feld des alten Objekts im *from-space*.
  Setze *scan* auf den Anfang des *to-space*.
- Solange $scan < next$: Verarbeite das Objekt an der Stelle *scan* und erhöhe danach *scan* um dessen Länge:
  - Aktualisieren aller Zeiger in den *from-space*, die auf ein bereits kopiertes Objekt verweisen (d.h. der erste Feldeintrag ist Forward-Pointer)
  - Kopieren aller anderen Kindobjekte in den *to-space*; eintragen von Forward-Pointern in die Originalobjekte.

Copying Collection — Cheney Algorithmus

Der Cheney-Algorithmus implementiert Breitensuche.

Das hat den Nachteil, dass nahe zusammenliegende Objekte nach einer Kopieroperation oft weit auseinander liegen können.

$\Rightarrow$ schlechte Cache-Performance

Abhilfe: Kombination von Breitensuche und Tiefensuche. Beim Kopieren eines Objekts verfolgt man einen Pfad von erreichbaren (und noch nicht kopierten) Objekten und kopiert diese gleich mit.

Generational Garbage Collection

Die Kosten eines kopierenden Garbage Collectors werden in der Praxis dadurch bestimmt, dass Objekte mit langer Lebensdauer immer wieder kopiert werden müssen.

Heuristik: Die meisten Objekte haben nur eine sehr kurze Lebensdauer, nur ein kleiner Anteil lebt lange.

Teile Speicher in Objekte verschiedener Generation auf. Der Bereich der junge Generationen wird oft vom Garbage Collector behandelt. Ältere Generationen entsprechend seltener.

- Da meist wird nur die junge Generation betrachtet wird, sind die GC-Pausen üblicherweise kürzer als bei vollständigem GC.
- Für ältere Objekte können auch andere Verfahren wie Mark-and-Sweep/Compact verwendet werden. Das reduziert den Gesamtspeicherplatzbedarf.

Generational GC wird in den meisten state-of-the-art Compilern verwendet (z.B. für Hotspot JVM, Haskell, OCaml).

Inkrementelle Garbage Collection

Der Programmablauf zerfällt in zwei „Prozesse“:

Kollektor: Er sammelt unerreichbare Objekte auf, führt also die eigentliche GC durch.

Mutator: Er fügt neue Objekte hinzu und verändert Referenzen; er entspricht dem eigentlichen Programm.

Bei den bisherigen Verfahren führt der Kollektor immer einen ganzen Ablauf durch; der Mutator wird während dieser Zeit angehalten (stop-the-world GC).

Bei nebenläufiger GC darf der Mutator auch während der Arbeit des Kollektors tätig werden, muss dabei aber mit diesem in geeigneter Weise kooperieren.

Abstraktion der GC-Algorithmen

Alle bisherigen GC-Algorithmen unterteilen die Objekte in drei Klassen:

- *Weiße Objekte*: Noch gar nicht besucht.
- *Graue Objekte*: Schon besucht, Kinder aber noch nicht vollst. verarbeitet (auf Stack, bzw. zwischen *scan* und *next*).
- *Schwarze Objekte*: Schon besucht, Kinder grau oder schwarz.

Grundalgorithmus:

- Manche Wurzelobjekte grau.
- Solange es graue Objekte gibt: Wähle graues Objekt, mache es schwarz und seine weißen Kinder grau.
- Wenn es keine grauen Objekte mehr gibt, können alle weißen gelöscht werden.

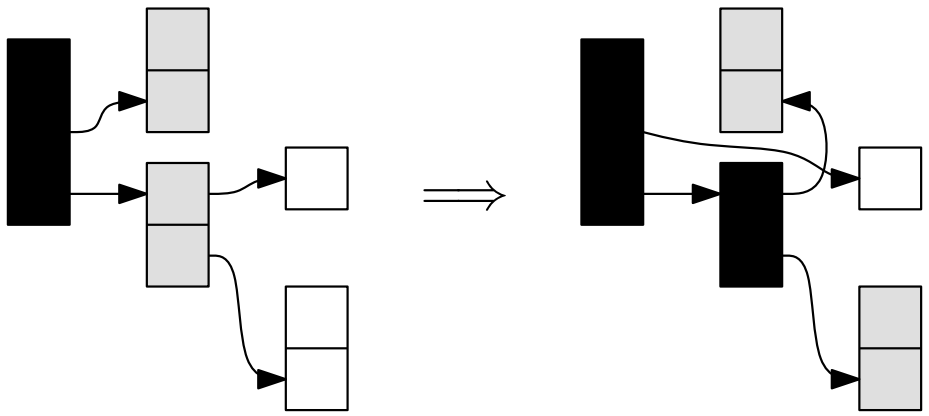
Abstraktion der GC-Algorithmen

Invarianten:

1. Schwarze Objekte verweisen nie auf weiße.
2. Graue Objekte befinden sich in einer Datenstruktur des Kollektors (grey set).

Die Invarianten stellen sicher: Wenn es keine grauen Objekte mehr gibt, dann können alle weißen sicher entfernt werden.

Beispiel: Wenn der Mutator die erste Invariante verletzt, könnten weiße Objekte fälschlicherweise freigegeben werden.



Konkrete Verfahren

- *Dijkstra, Lamport, et al.*: Will der Mutator ein weißes Objekt in ein Feld eines schwarzen Objekts schreiben, so ist ersteres grau zu machen (durch zusätzliche Instruktionen bei einer write-Operation; **write barrier**).
- *Steele*: Will der Mutator ein weißes Objekt in ein Feld eines schwarzen Objekts schreiben, so ist letzteres grau zu färben (durch zusätzliche Instruktionen bei einer write-Operation; **write barrier**).
- *Baker*: Referenziert der Mutator ein weißes Objekt, egal zu welchem Zweck, so wird dieses sofort grau gefärbt (**read barrier**).

Baker Algorithm

Bakers Algorithm für inkrementelles GC

- Basiert auf Cheneys Algorithmus für Copying GC.
- Der Start einer GC erfordert nur das „forwarding“ der Roots.
- Mit jeder Allokierung wird die Scan-Operation etwas weitergeführt.
- Allokiert wird vom Ende des to-spaces an abwärts, in Richtung `next`-Pointer. Neue Objekte sind damit automatisch schwarz.
- **to-space** Invariante: Der Mutator sieht nur Pointer in den to-space; versucht er einen Pointer in den from-space zu folgen, so wird dieses Objekt sofort in den to-space kopiert und damit grau gefärbt. Dies erfordert eine **read barrier**.
- Kosten dominiert von den Checks für die read-barrier.

Weiterführendes

- **Page-based GC**: Viele Verfahren, insbes. nebenläufige können auf der Ebene von Seiten des virtuellen Speichers implementiert werden. Dabei kann spezielle OS-Infrastruktur, z.B. Read-only Bits benutzt werden.
- **Realtime GC**: Techniken für Echtzeit GC sind inzwischen soweit ausgereift, dass sie in Real-Time Java verwendet werden.

Garbage Collection im Compiler

Der Garbage Collector muss Zeiger in den Heap von anderen Daten unterscheiden können und auch die Wurzelobjekte kennen.

Typsichere Sprachen

- Die Felder von Objekten können zum Beispiel anhand der Klassen-Id als Zeiger identifiziert werden (in Java).
- OCaml benutzt das letzte Bit (0 für Pointer), Einschränkung auf 31-Bit Integer.
- Objekte können ohne Konsequenzen verschoben werden.
- Wurzelobjekte können statisch vom Compiler erkannt werden.

Garbage Collection im Compiler

Wurzelobjekte im MiniJava Compiler:

- Durch konsequentes Mitführen von Typinformationen (Zeiger oder Skalar) in der Zwischensprache (z.B. Temporaries) und im `Frame` (z.B. in `allocLocal`), können wir für jeden Programmpunkt angeben, welche Register und Stackzellen Zeiger enthalten.
- Da Garbage Collection nur bei Funktionsaufrufen passieren kann, müssen die Wurzelobjekte nur dann bekannt sein.
- Der Compiler legt für jeden CALL-Programmpunkt eine *Pointer Map* in einem Datenfragment ab.
- Daraus kann der Garbage Collector die Menge Wurzelobjekte konstruieren: Durch Ablaufen des Stack werden die Adressen aller noch nicht beendeter Funktionsaufrufe berechnet. Diese Adressen werden als Index benutzt, um die Pointer Maps der einzelnen Aufrufe zu erfragen. Damit können Zeiger identifiziert werden.

Garbage Collection für unsichere Sprachen

Unsichere Sprachen wie C/C++:

- Zeiger sind nicht abstrakt, Objekte können i.a. nicht verschoben werden
- Wurzelobjekte können nicht statisch berechnet werden

Conservative Garbage Collectors (z.B. Boehm, Demers, Weiser)

- benutzen Heuristiken, um Garbage Collection für (eine Teilmenge von) unsicheren Sprachen verfügbar zu machen
- Es wird zur Laufzeit versucht zu erkennen, welche Werte in Registern, Stack, ... Zeiger sein könnten. Alle Objekte im Speicher, die so referenziert werden, werden erhalten.
- Diese Kollektoren sind konservativ, d.h. sie behalten oft mehr im Speicher als nötig, können aber natürlich nicht mit beliebiger Zeigerarithmetik umgehen.

Garbage Collection für unsichere Sprachen

Eine Boehm-Demers-Weiser Conservative GC (`libgc`) kann auch (temporär) in der MiniJava-Runtime verwendet werden.

Die Laufzeitbibliothek muss lediglich folgendermaßen angepasst werden:

```
#include <gc/gc.h>
int32_t L_halloc(int32_t size) {
  return (int32_t)GC_MALLOC(size);
}
```

(Kompilation dann mit `gcc prg.s runtime.c -lgc`)

Bemerkung: Das funktioniert natürlich nur, wenn der Garbage Collector die Wurzelobjekte stets richtig erkennt, d.h. wenn der MiniJava-Compiler nur beschränkt Zeigerarithmetik benutzt. Insbesondere Optimierungen können später problematisch werden.

Optionale Programmieraufgabe

Implementierung eines Mark-and-Sweep Garbage Collectors für MiniJava, bzw. Integration einer Bibliothek für Garbage Collection.

Dies erfordert:

- Klassen- und Array-Deskriptoren mit Pointer Information: Zur Laufzeit wird für jede Klasse ein Deskriptor benötigt, der beschreibt welche Felder Zeiger sind.
- Pointer-map für alle abstrakten Register und alle Variablen auf dem Stack: eine Abbildung von Registern/Variablen auf Boolean (Pointer oder kein Pointer).
- Einbindung der Pointer maps als „data fragments“ in den Code nach jedem Funktionsaufruf.
- Im Laufzeitsystem: Implementierung von Mark und Sweep, z.B. mit DFS