

Compiler Construction 2024: Garbage collection

Annette Bieniusa, Konrad Anton

May 28, 2024

Outline

- 1 Introduction
- 2 Reference counting
- 3 Mark-and-Sweep
- 4 Copying Collection
- 5 Generational Collection
- 6 Incremental and Concurrent Collection
- 7 Integration with compiler

Types of storage

Static allocation

- All names in the program are bound to a storage location known at compile-time
- Very fast due to direct access
- Safe as the program cannot run out of memory
- Drawback: recursion not possible

Stack allocation (procedure local data)

- Stored in an activation record/frame
- Values do not persist from one activation to next
- Size may depend on parameters passed to procedure
- Only objects whose size is known at compile time can be returned by a procedure

Heap allocation

- Data allocation and deallocation independent from program flow
- Size of data structures may vary dynamically
- Dynamically-sized objects can be returned by procedure
- Required for recursive data structures (lists, trees, etc)

Manual memory management

- API for allocation and deallocation, e.g., for C
 - `malloc (size)` — returns a pointer to an unused, contiguous record of memory of demanded size
 - `free (record)` — declares that the record is no longer used and can be reclaimed
 - manages a `freelist` that contains unused records of different sizes; allocation takes a record from the `freelist` and splits it to obtain one of demanded size; deallocation returns the record to the `freelist`
- Advantages: flexible, application specific policies, semantic deallocation, efficient
- Disadvantages: error prone, memory leaks, premature deallocation, complicated reasoning

Management of dynamically allocated storage

Automatic memory management — Garbage Collection

- API only provides allocation; deallocation is automatic
- Goal: reclaim unused records as early as possible
- Advantages: no user/programmer interaction for deallocation required, no premature deallocation (safety)
- Disadvantages: extra time needed for memory management, deallocation based on reachability \Rightarrow memory leaks

Terminology

- mutator = user program
- collector = memory management agent

Reachability

- Program variables and heap-allocated records form a directed graphs
- Local and global variables are roots of this graph

Reachability

A record in the heap is *reachable* if its address is held in a root, or there is a pointer to it held in another live heap record.

$$\mathit{reach} = \{n \in \mathit{Records} \mid (\exists r \in \mathit{Roots} : r \rightarrow n) \vee (\exists m \in \mathit{reach} : m \rightarrow n)\}$$

- Requirement: no random access to locations in address space — the program only points to previously allocated records
- (safe) approximation

Outline

- 1 Introduction
- 2 Reference counting**
- 3 Mark-and-Sweep
- 4 Copying Collection
- 5 Generational Collection
- 6 Incremental and Concurrent Collection
- 7 Integration with compiler

Reference counting

Idea: keep track during execution how many pointers to a record exist!

For each access $y \leftarrow p$

```
1 z <- y
2 z.count <- z.count-1
3 if z.count=0
4     putOnFreelist(z)
5 y <- p
6 p.count <- p.count+1
```

```
1 function putOnFreeList(p)
2     for all fields f_i of p
3         p.f_i.count <- p.f_i.count-1
4         if p.f_i.count=0 putOnFreelist(p.f_i)
5     p.f_l <- freelist
6     freelist <- p
```

Advantages

- Predictable
- No need to know all roots
- GC effort spread over run time, no pauses

Problems

- Cycles of garbage cannot be reclaimed
 - Require programmer to break cycles explicitly
 - Combine reference counting with occasional mark-and-sweep
- Counters are expensive
 - Aggregate changes to counters via data flow analysis
 - Static analysis to avoid changes to counters
- Complex memory management code at every pointer update

Outline

- 1 Introduction
- 2 Reference counting
- 3 Mark-and-Sweep**
- 4 Copying Collection
- 5 Generational Collection
- 6 Incremental and Concurrent Collection
- 7 Integration with compiler

Mark-and-Sweep Collection

- Global traversal of all reachable objects to determine which ones maybe reclaimed
- Only started when available storage is exhausted
- Depth-first search marks all reachable nodes
- `freelist` contains pointers to available storage

Mark phase

```
1 for each root v
2   DFS(v)
3
4 function DFS(x)
5   if x is pointer into heap to record p
6     if record p is not marked
7       mark p
8     for each field f_i of record p
9       DFS(p.f_i)
```

Sweep phase

```
1 p <- first address in heap
2 while p < last address in heap
3   if record p is marked
4     unmark
5   else let f_1 be the first field in p
6     p.f_1 <- freelist
7     freelist <- p
8   p <- p + (size of record p)
```

- R = words of reachable data
- H = size of heap

Analysis

- Mark phase: $c_1 R$
- Sweep phase: $c_2 H$
- Regained memory: $H - R$
- Amortized cost:

$$\frac{c_1 R + c_2 H}{H - R}$$

Worst case (for M&S)

Heap is filled with one long linked list. Calls to DFS nested $\Omega(H)$ deep!

Countermeasures:

- Emergency stop at full stack, then search heap for marked nodes with unmarked children
- Pointer reversal
 - While visiting y coming from t via $x.f$, use $x.f$ to point *back* to t .
 - DFS stack hidden in heap
 - Needs field `done` for each record

Pointer reversal

```
1 function DFS(x)
2   if x is a pointer and record x is not marked
3     t <- nil
4     mark x; done[x] = 0
5     while true
6       i <- done[x]
7       if i < number of fields in record x
8         y <- x.f_i      // index starts at 0
9         if y is a pointer and record y not marked
10          x.f_i <- t; t <- x; x <- y
11          mark x; done[x] = 0
12        else
13          done[x] <- i+1
14        else           // back to parent!
15          y <- x; x <- t
16          if x = nil then return
17          i <- done[x]
18          t <- x.f_i; x.f_i <- y
19          done[x] <- i+1
```

- Organizing the freelist
 - Array of several freelists
 - `freelist[i]` points to linked list of all records of size `i`
 - If `freelist[i]` is empty, grab entry from `freelist[j]` ($j > i$) putting unused portion back to `freelist[j-i]`
- Fragmentation

Outline

- 1 Introduction
- 2 Reference counting
- 3 Mark-and-Sweep
- 4 Copying Collection**
- 5 Generational Collection
- 6 Incremental and Concurrent Collection
- 7 Integration with compiler

Copying collection

- Idea: build an isomorphic, compact image of the heap
 - Partition heap into from-heap and to-heap
 - Allocate data in from-heap via `free`-pointer
 - Garbage collector moves all reachable data to to-heap
 - Everything left is garbage
 - Reverse role of to-heap and from-heap
- To-space copy is compact \Rightarrow no fragmentation
- Simple allocation: add requested size to `free`-pointer.

Breadth-first copying

```
1 scan <- free <- beginning of to-space
2 for each root r
3   r <- Forward(r)
4 while scan < free
5   for each field f_i of record at scan
6     scan.f_i <- Forward(scan.f_i)
7   scan <- scan + (size of record at scan)
```

Forwarding a pointer

```
1 function Forward(p)
2   if p points to from-space
3     then if p[0] points to to-space
4       then return p[0]
5       else for each field f_i of p
6         free.f_i <- p.f_i
7         p[0] <- free
8         free <- free + (size of record p)
9         return p[0]
10  else return p
```

Locality of references

- Records that are copied near each other have the same distance from the roots
- If record p points to record s , they will likely be far apart
⇒ bad caching behavior
- But: depth-first copying requires pointer-traversal
- hybrid solution: use breadth-first copying, but take direct children into account

Locality of references

```
1 function Forward(p)
2   if p points to from-space
3     then if p[0] points to to-space
4         then return p[0]
5         else Chase(p); return p[0]
6     else return p
7
8 function Chase(p)
9   repeat
10    q <- free           // q is the new p
11    free <- free + (size of record p)
12    r <- nil           // some child of p to copy along
13    for each field f_i of record p
14      q.f_i <- p.f_i
15      if q.f_i points to from-space
16          and q.f_i[0] does not point to to-space
17        then r <- q.f_i
18    p[0] <- q
19    p <- r
20 until p = nil
```


Analysis

- Breadth-first search: $O(R)$
- Regained memory: $H/2 - R$
- Amortized cost:

$$\frac{c_3 R}{\frac{H}{2} - R}$$

- Realistic setting: $H = 4R$
- high costs for copying! $c_3 \gg c_2, c_1$.

Outline

- 1 Introduction
- 2 Reference counting
- 3 Mark-and-Sweep
- 4 Copying Collection
- 5 Generational Collection**
- 6 Incremental and Concurrent Collection
- 7 Integration with compiler

Generational Collection

- Hypothesis: a newly created object is likely to die soon (*infant mortality*); if it survived several collection cycles, it is likely to survive longer
- Idea: collector concentrates on younger data
- Divide the heap into *generations*
- G_0 contains the most recently allocated data, G_1, G_2, \dots contain older objects
- Enlarge the set of roots to also include pointers from $G_1, G_2 \dots$ to G_0 :
 - need to track updating of fields
 - use a *remembered list/set* to collect updated objects and scan this for root pointers at garbage collection

Generational Collection

- Use same system to garbage collect also older generations.
- Move objects from G_i to G_{i+1} after several collections.
- Possible to use the virtual memory system:
 - Updating an old generation sets a dirty bit for the corresponding page
 - If OS does not make dirty bits available, the user program can use write-protection for the page and implement user-mode fault handler for protection violations

Generational Collection

Tuning parameters:

- Number of generations
- Relative size of generations
- Promotion threshold

Outline

- 1 Introduction
- 2 Reference counting
- 3 Mark-and-Sweep
- 4 Copying Collection
- 5 Generational Collection
- 6 Incremental and Concurrent Collection**
- 7 Integration with compiler

Incremental and concurrent collection

- Collector might interrupt the program for a long time
- Undesirable for interactive or real-time programs
- Idea: Perform GC in small increments

Incremental collection: collector performs only part of a collection on each allocation

Concurrent collection: collector and mutator(s) run in parallel

Tri-Color marking

White objects have not yet been visited.

Grey have been visited, but their children not yet.

Black have been visited as well as their children.

Basic algorithm

```
1 color all objects white
2 for each root r
3   if r points to an object p
4     color p grey
5 while there are any grey objects
6   select a grey record p
7   for each field f_i of p
8     if record p.f_i is white
9       color record p.f_i grey
10  color record p black
```


Invariants

- 1 No black object points to a white object.
 - 2 Every grey object is on the collector's (stack or queue) data structure.
- Mutator must not violate these invariants.
 - Synchronization of mutator and collector is necessary.

The big danger

- Treating garbage as possibly reachable: acceptable
- Treating reachable data as garbage: bad! Happens only if:
 - 1 Mutator stores pointer to white *a* into black object, and
 - 2 the original reference to *a* is destroyed

Write-barrier Algorithms

Goal: fix invariant violations whenever the mutator stores pointers to white objects.

Possible approaches:

- Whenever the mutator stores a pointer to white a into a black object b , it colors a grey. (\Rightarrow a reachable)
- Whenever the mutator stores a pointer to white a into a black object b , it colors b grey. (\Rightarrow check b again)
- Use paging
 - Mark all-black pages as read-only
 - When mutator writes into all-black object, page fault!
 - Page fault handler colors all objects on the page grey.

Read-barrier Algorithms

Ensure that the mutator never sees a white object.

- Whenever the mutator fetches a pointer b to a white object, it colors b grey.
- Use paging
 - Invariant: mutator only sees black objects
 - Goal: whenever mutator loads a non-black object, scan it and children
 - Use page protection to trap reads to pages containing white or grey objects
 - Page fault handler scans the page until black

Baker's Algorithm

- When starting new gc cycle: Flip
 - 1 Swap roles of from-space and to-space.
 - 2 Forward all roots to to-space.
 - 3 Resume mutator.
- For each allocation:
 - 1 Scan a few pointers at scan.
 - 2 Allocate new record at the end of to-space.
 - 3 When scan reaches free, terminate gc for this cycle.
- For each fetch:
 - 1 Check if fetched pointer points to from-space.
 - 2 If so, forward pointed immediately. (Mutator never sees white objects)

Outline

- 1 Introduction
- 2 Reference counting
- 3 Mark-and-Sweep
- 4 Copying Collection
- 5 Generational Collection
- 6 Incremental and Concurrent Collection
- 7 Integration with compiler**

Compiler interacts with GC by

- generating code for allocating data
- describing locations of roots
- describing data layout on heap
- implementing read/write barriers

Example: Allocating record of size N when using copying collection:

- 1 Call the allocate function.
- 2 Test $free + N < limit?$ \Rightarrow If not, call gc.
- 3 Move free into result
- 4 Clear memory locations $free, \dots, free+N-1$
- 5 $free \leftarrow free + N$
- 6 Move result into required place.
- 7 Store values into the record.

Fast Allocation

How much data is allocated on average?

- approximately one word of allocation per store instruction
- 1/7 of all instructions are stores

Possible optimization:

- Inline the allocate function.
- Move result directly into the right register.
- Combine clearing and initialization of fields.
- Allocate data for a whole block to minimize tests.

Data layouts

- Save for every heap object a pointer to its class-/type-descriptor
 - What is the total size of this object?
 - Which fields are pointers?
 - (For dynamic method lookup: vtable)
- Save all pointer-containing temporaries and local variables in a pointer map
 - different at every program point \Rightarrow save it only at calls to alloc and function calls
 - Collector starts at top of stack and scans all frames, handling all the pointers in that frame as saved in the pointer-map entry for this frame
 - Information about callee-save registers needs to be transferred to callee.

- Jones, R. and Lins, R. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Chichester, England (1996).