Compiler Construction 2009/2010: Garbage collection

Annette Bieniusa

January 18, 2010
Types of storage

Static allocation
- All names in the program are bound to a storage location known at compile-time
- does not allow recursion
- very fast due to direct access
- safe as the program cannot run out of memory

Stack allocation
- Local data is stored in an activation record/frame
- Values do not persist from one activation to next
- Size of local data may depend on parameters passed to procedure
- Only objects whose size is known at compile-time can be returned by a procedure
Types of storage

Heap allocation

- Data may be allocated (and deallocated) in any order
- Size of data structure can be varied dynamically
- Dynamically-sized objects and closures (i.e. function paired with environment) can be returned by procedure
- Allows recursive data structures
What is garbage collection?

Garbage collection

- automatic management of dynamically allocated storage
- performed at run-time

Terminology

- mutator = user program
Reachability

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

Liveness

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

\[
\text{live} = \{ n \in \text{Records} \mid (\exists r \in \text{Roots} : r \rightarrow n) \lor (\exists m \in \text{live} : m \rightarrow n) \}\]

- Requirement: no random access to locations in address space.
- (safe) approximation.
Outline

1. Introduction
   - Types of storage

2. Reference counts

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 $x.f_i \leftarrow p$

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

function putOnFreeList(p)
    for all fields f_i of p
        p.f_i.count <- p.f_i.count-1
        if p.f_i.count=0 putOnFreelist(p.f_i)
    p.f_1 <- freelist
    freelist <- p
```
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
Mark-and-Sweep Collection

- global traversal of all live 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
for each root v
  DSF(v)

function DFS(x)
  if x is pointer into heap to record p
    if record p is not marked
      mark p
      for each field f_i of record p
        DFS(p.f_i)
Algorithm

Sweep phase

1. $p \leftarrow$ 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. $p.f_1 \leftarrow$ freelist
      7. freelist $\leftarrow p$
      8. $p \leftarrow p +$ (size of record $p$)
Costs

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

Analysis

- Mark phase: $O(R)$
- Sweep phase: $O(H)$
- Regained memory: $H - R$
- Amortized cost:
  \[ \frac{c_1 R + c_2 H}{H - R} \]
function DSF(x)
    if x is a pointer and record x is not marked
        t <- nil
        mark x; done[x] = 0
    while true
        i <- done[x]
        if i < number of fields in record x
            y <- x.f_i
            if y is a pointer and record y not marked
                x.f_i <- t; t <- x; x <- y
                mark x; done[x] = 0
            else
                done[x] <- i+1
        else
            y <- x; x <- t
            if x = nil then return
            i <- done[x]
            t <- x.f_i; x.f_i <- y
            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
   - Types of storage
2. Reference counts
3. Mark-and-Sweep
4. Copying Collection
5. Generational Collection
6. Incremental and Concurrent Collection
7. Integration with compiler
Idea: build an isomorphic, compact image of the heap
- Partition heap into from-heap and to-heap
- Use from-heap to allocate data
- When invoking garbage collection, move 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
Cheney’s Algorithm

Breadth-first copying

1. scan <- next <- beginning of to-space
2. for each root r
   r <- Forward(r)
3. while scan < next
   for each field f_i of record at scan
     scan.f_i <- Forward(scan.f_i)
4. scan <- scan + (size of record at scan)
Cheney’s Algorithm

**Forwarding a pointer**

```python
function Forward(p)
    if p points to from-space
        then if p.f_1 points to to-space
            then return p.f_1
        else for each field f_i of p
            next.f_i <- p.f_i
            p.f_1 <- next
        next <- next + (size of record p)
        return p.f_1
    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 $\Rightarrow$ 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

function Forward(p)
  if p points to from-space
    then if p.f_1 points to to-space
      then return p.f_1
    else Chase(p); return p.f_1
  else return p

function Chase(p)
  repeat
    q <- next
    next <- next + (size of record p)
    r <- nil
    for each field f_i of record p
      q.f_i <- p.f_i
      if q.f_i points to from-space
        and q.f_i.f_1 does not point to to-space
        then r <- q.f_i
      p.f_1 <- q
      p <- r
  until p = nil
### Costs

**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!
Outline

1. Introduction
   - Types of storage
2. Reference counts
3. Mark-and-Sweep
4. Copying Collection
5. Generational Collection
6. Incremental and Concurrent Collection
7. Integration with compiler
Generational Collection

- Hypothesis: newly created objects are 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, \( G_1, G_2, \ldots \) contain older objects
- Enlarge the set of roots to also include pointers from \( G_1, G_2 \ldots \) 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.
- Each older generation should be exponentially bigger than the previous one.
- 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
Incremental and concurrent collection

- Collector might interrupt the program for a long time
- undesirable for interactive or real-time programs
- Idea: interleave gc work with program execution

**Incremental collection:** collector only operates when mutator requires it

**Concurrent collection:** collector operates between or during the program execution
Tricolor marking

**White** objects are not yet visited.

**Grey** are visited, but their children not yet.

**Black** are marked as well as their children.

---

**Basic algorithm**

```plaintext
color all objects white
for each root r
  if r points to an object p
    color p grey
while there are any grey objects
  select a grey record p
  for each field f_i of p
    if record p.f_i is white
      color record p.f_i grey
  color record p black
```
Tricolor marking

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.
Whenever the mutator stores a white pointer $a$ into a black object $b$, it colors $a$ grey. (⇒ $a$ reachable)

Whenever the mutator stores a white pointer into a black object, it colors $b$ grey. (⇒ check $b$ again)
All-black pages are marked read-only in the memory system. Whenever the mutator stores any value into an all-black page, a page fault marks all objects on that page grey.

Whenever the mutator fetches a pointer b to a white object, it colors b grey.

Whenever the mutator fetches a pointer b from any memory page containing a non-black object, a page-fault handler colors every object on the page black (making children of these objects grey).
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 next, terminate gc for this cycle.

- For each fetch:
  1. Check if fetched pointer points to from-space.
  2. If so, forward pointed immediately.
Outline

1 Introduction
   - Types of storage
2 Reference counts
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
Fast allocation

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

1. Call the allocate function.
3. Move next into result
4. Clear memory locations next, ..., next+N-1
5. next <- next + 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 ⇒ 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 need to be transferred to callee.