Compiler Construction

Garbage collection

University of Freiburg

Annette Bieniusa, Konrad Anton, *Matthias Keil*

University of Freiburg

30. Januar 2017

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

University of Freiburg

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)

Stack vs. Heap Allocation

Stack allocation

- Fast access
- No explicit de-allocate required
- No fragmentation (efficient space management)
- Local variables only
- Limit on stack size

Heap allocation

- Global variables
- No limit on memory size
- Slower access
- Memory become fragmented over time

Management of dynamically allocated storage

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

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

- 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.

$$\begin{aligned} reach = \{ n \in Records \mid & \quad (\exists r \in Roots : r \rightarrow n) \\ \vee & \quad (\exists m \in reach : m \rightarrow n) \} \end{aligned}$$

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

Reference counting

University of Freiburg

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

For each access `y <- p`

```
z <- y
z.count <- z.count-1
if z.count=0
  putOnFreelist(z)
  y <- p
  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
```

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
- Complex memory management code at every pointer update

Mark-and-Sweep Collection

University of Freiburg

- 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

UNI FREIBURG

Mark phase

```
for each root v
  DFS(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)
```

Sweep phase

```
p <- first address in heap
while p < last address in heap
  if record p is marked
    unmark
  else let f_1 be the first field in p
    p.f_1 <- freelist
    freelist <- p
  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

```
function DFS(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      // index starts at 0
          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                // back to parent!
          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

Copying collection

- 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
- Simple allocation: add requested size to `next`-pointer.

Cheney's Algorithm

University of Freiburg

Breadth-first copying

```
scan <- next <- beginning of to-space
for each root r
  r <- Forward(r)
while scan < next
  for each field f_i of record at scan
    scan.f_i <- Forward(scan.f_i)
  scan <- scan + (size of record at scan)
```

Forwarding a pointer

```
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

```
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              // q is the new p
    next <- next + (size of record p)
    r <- nil     // some child of p to copy along
    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
```

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! $c_3 \gg c_2, c_1$.

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, $G_1, G_2, \ldots$ 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.
- 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

Tuning parameters:

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

- 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

University of Freiburg

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

```
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
```

Tri-Color marking

University of Freiburg

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

University of Freiburg

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.

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 next, 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)

Interface to the 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 `next + N < limit?` $\Rightarrow$ If not, call gc.
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.

- 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 transfered to callee.

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