Automatic Garbage Collection

Automatically free dead objects
- no **dangling pointers**, no **storage leaks** (maybe)
- can have faster allocation, better memory locality

General styles:
- reference counting
- tracing
  - mark/sweep, mark/compact
  - copying

Adjectives:
- generational
- conservative
- incremental
- parallel
- distributed

Reference counting

For each heap-allocated object,
maintain count of # of pointers to object
- when create object, ref count = 0
- when create new ref to object, increment ref count
- when remove ref to object, decrement ref count
- if ref count goes to zero, then delete object

```
proc foo() {
  let a := new Cons;
  let b := new Blob;
  return bar(a, b);
}

proc bar(x, y) {
  let l := x;
  l.head := y;
  return l.tail;
}
```


Evaluation

+ local, incremental work
+ little/no language support required
+ local ⇒ feasible for distributed systems

– cannot reclaim cyclic structures
– heap gets fragmented
– high run-time overhead (10-20%)
  - deferred ref counting [Deutsch & Bobrow 76]
– space cost

Tracing collectors

Start with a set of **root** pointers
- global vars
- contents of stack & registers

Traverse objects transitively from roots
- visits **reachable** objects
- all unvisited objects are garbage

Issues:
- how to identify pointers?
- what order to visit objects?
- how to know an object is visited?
- how to free unvisited objects?
- how to synchronize collector and program (**mutator**)?

Identifying pointers

"Accurate":
- static type info & compiler support
- run-time tagging scheme
- run-time conventions about where pointers can be

Conservative [Bartlett 88, Boehm & Weiser 88]:
- assume anything that looks like a pointer is a pointer
  - most optimistic: just aligned pointers to beginning of objects
  - interior "pointers"?
  - unaligned "pointers"?

Mark/sweep collectors

[McCarthy 60]: stop-the-world tracing collector

Stop the application when heap fills

Trace reachable objects
- set mark bit in each object
- tracing control:
  - depth-first, recursively using separate stack
  - depth-first, using pointer reversal

Sweep through all of memory
- add unmarked objects to free list
- clear marks of marked objects

Restart mutator

Evaluation

+ collects cyclic structures
+ simple to implement

– "embarrassing pause" problem
– poor memory locality
  - when tracing
  - when sweeping
  - fragments the heap
– not suitable for distributed systems

Some improvements

Mark/compact collection:
when sweeping through memory, compact rather than free
+ reduces fragmentation
+ allocation can be very fast
– slower to sweep
– can change pointers
⇒ requires accurate info about pointers

Generational mark/∗

Incremental, parallel mark/∗
+ reduce embarrassing pause problem
– more complex

Copying collectors

[Cheney 70]

Divide heap into two equal-sized **semi-spaces**

- mutator allocates in **from-space**
- **to-space** is empty

When from-space fills, do a GC:

- visit objects referenced by roots
- when visit object:
  - copy to to-space
  - leave forwarding pointer in from-space version
  - if visit object again, just redirect pointer to to-space copy
- scan to-space linearly to visit reachable objects
  - to-space acts like breadth-first-search work list
- when done scanning to-space:
  - empty from-space
  - **flip**: swap roles of to-space and from-space
- restart mutator

Evaluation

+ collects cyclic structures
+ supports compaction, fast allocation automatically
+ no separate traversal stack required
+ only visits reachable objects, not all objects

– requires twice the (virtual) memory, physical memory sloshes back and forth
  - OS support?
– "embarrassing pause" problem still
– copying can be slow
– changes pointers

An improvement

Add **eden** (a.k.a. **nursery**) semi-space [Ungar 84]

- eden-space fits in main memory (or cache)
- mutator allocates in eden-space
- GC when eden-space fills
  - copy eden-space + from-space to to-space
  - flip: empty both eden-space and from-space

+ reduces cache misses, page faults
  - most heap memory references satisfied in eden-space?

– eden-space + from-space can overflow to-space
– more complex

Another improvement

Add semi-space for large objects [Caudill & Wirfs-Brock 86]

- big objects slow to copy, so allocate them in separate space
- use mark/sweep in large object space

+ no copying of big objects
– more complex

Generational GC

Observation:
most objects die soon after allocation
- e.g. closures, cons cells, stack frames, numbers

Idea:
concentrate GC effort on young objects
- divide up heap into 2 or more generations
- GC each generation with different frequencies, algorithms

Original idea: Peter Deutsch
Generational mark/sweep: [Lieberman & Hewitt 83]
Generational copying GC: [Ungar 84]

Generation scavenging

[Ungar 84]

2 generations: **new-space** and **old-space**
- new-space managed as a 3-space copying collector
- old-space managed using mark/sweep
- new-space much smaller than old-space

Apply copy collection to new-space frequently

If object survives many scavenges, then copy it to old-space
- **tenuring** (a.k.a. **promotion**)
- maintain age field in each object

If old-space (nearly) full, do a full GC

Roots for generational GC

Must include pointers from old-space to new-space as roots when scavenging new-space
How to find these?

Option 1: scan old-space at each scavenge

Option 2: track pointers from old-space to new-space

How to remember pointers?
- individual words containing pointers [Hosking & Moss 92]
- **remembered set** of objects possibly containing pointers [Ungar 84]
- **card marking** [Wilson 89]

How to update table?
- functional languages: easy!
- imperative languages: need a **write barrier**
  - specialized hardware
  - standard page protection hardware
  - in software, inserting extra checking code at stores

Evaluation

+ scavenges are short: fraction of a second
+ low run-time overhead
  - 2-3% in Smalltalk interpreter
  - 5-15% in optimized Self code
+ less VM space than pure copying
+ better memory locality than pure mark/sweep

– requires write barrier
– still have infrequent full GC's
– need space for age fields

Extensions

Multiple generations
- e.g. Ephemeral GC: 8 generations [Moon 84]
- may obviate need for age fields

Feedback-mediated tenuring policy [Ungar & Jackson 88]

Large object space

Incremental & parallel GC

Avoid long pause times by running collector & mutator in parallel
- physical or simulated parallelism

Main issue: how to synchronize collector & mutator?
- read barrier [Baker 78, Moon 84]
- write barrier [Dijkstra 78; Appel, Ellis & Li 88]