Garbage Collection

Weiyuan Li

Why GC exactly?

- Laziness
- Performance
  - `free` is not free
  - combats memory fragmentation
- More flame wars

Basic concepts

- Type Safety
  - Safe: ML, Java (not really)
  - Unsafe: C/C++
- Reachability
- Root set

Reference Counting GC

- Identifies garbage as an object changes from reachable to unreachable.
- Each object keeps a count. Once the count falls to zero, the object can be freed

Reference Counting GC (cont.)

- High overhead
  - additional operations
  - extra space
  - not evenly-distributed (so is manual memory management)
- Cannot handle self-referencing structures
  - no TSP for Perl
  - cycle detection (Python)

Reference Counting GC (cont.)

- Simple enough, works in most situations
  - Cyclic data structures are not that common
- One huge benefit
  - No more `close`/`closedir`

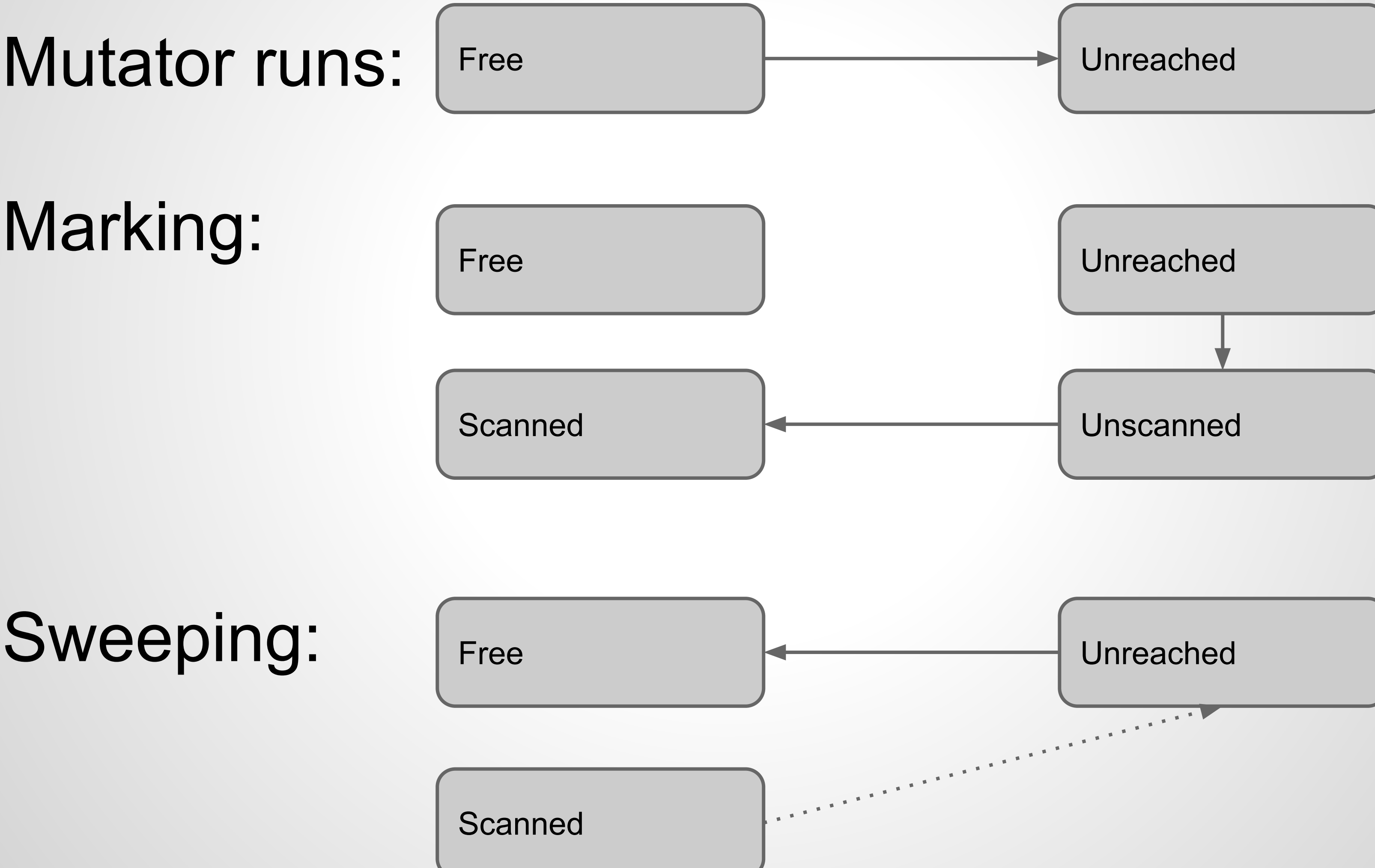
Trace-Based GC

- Runs periodically
- Starting from the root set, find all reachable objects and reclaim the rest
- Stop-the-world style

Mark-and-sweep

- Chunks are presumed unreachable, unless proven reachable by tracing
- Marking phase
- Sweeping phase

Free, Unreached, Unscanned, Scanned

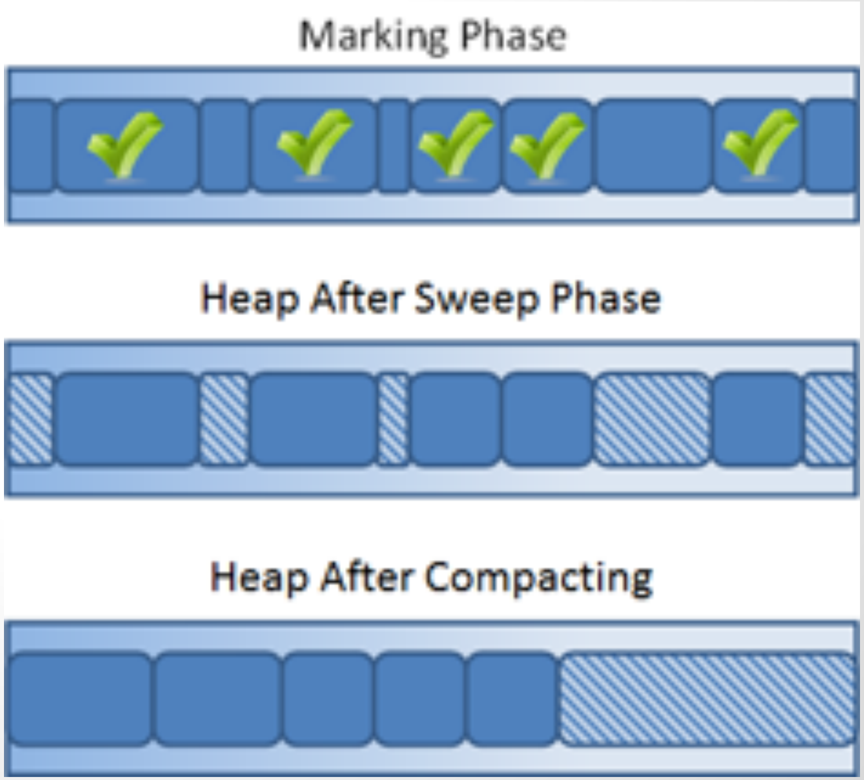


Baker’s mark-and-sweep GC

- Avoids examining the entire heap by maintaining a list of allocated objects (Unreached)
- Returns modified Free and Unreached lists

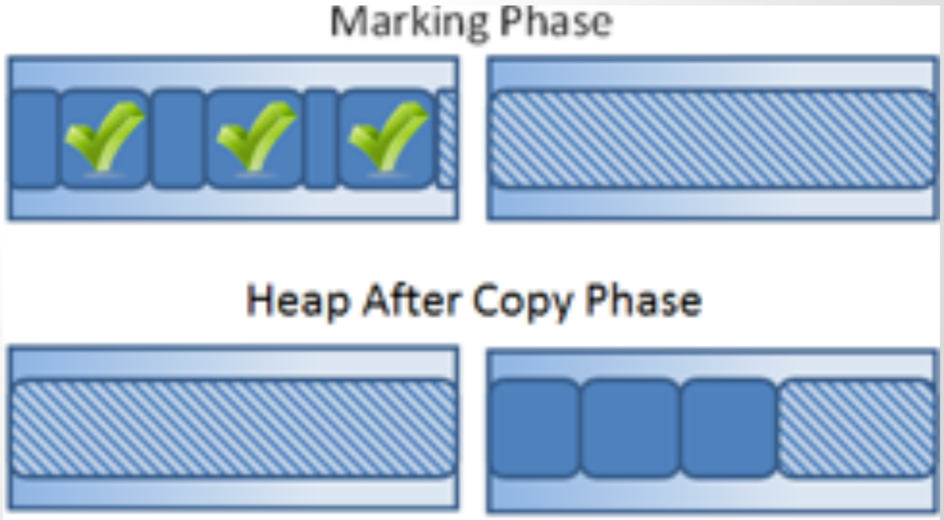
Mark-and-compact

- Moves reachable objects around to eliminate fragmentation
- Allocation is fast
- Better locality



Copying collector

- Divide the heap into two semispaces.
- Marking phase: find reachable objects
- Copy phase: copy all reachables the other semispace
- Improved: Cheney’s collector



Comparison

Basic Mark-and-sweep	# of memory chunks in heap
Baker’s algorithm	# of reached objects
Basic Mark-and-copy	# of chunks + reached objects
Cheney’s collector	# of reached objects

More...

- Adaptive collector
- Incremental garbage collection
- Partial-collection
  - Objects “die young”
  - Generational collector (copying partial-collection)
- The Train Algorithm
  - handles mature objects better

Boehm Garbage Collection

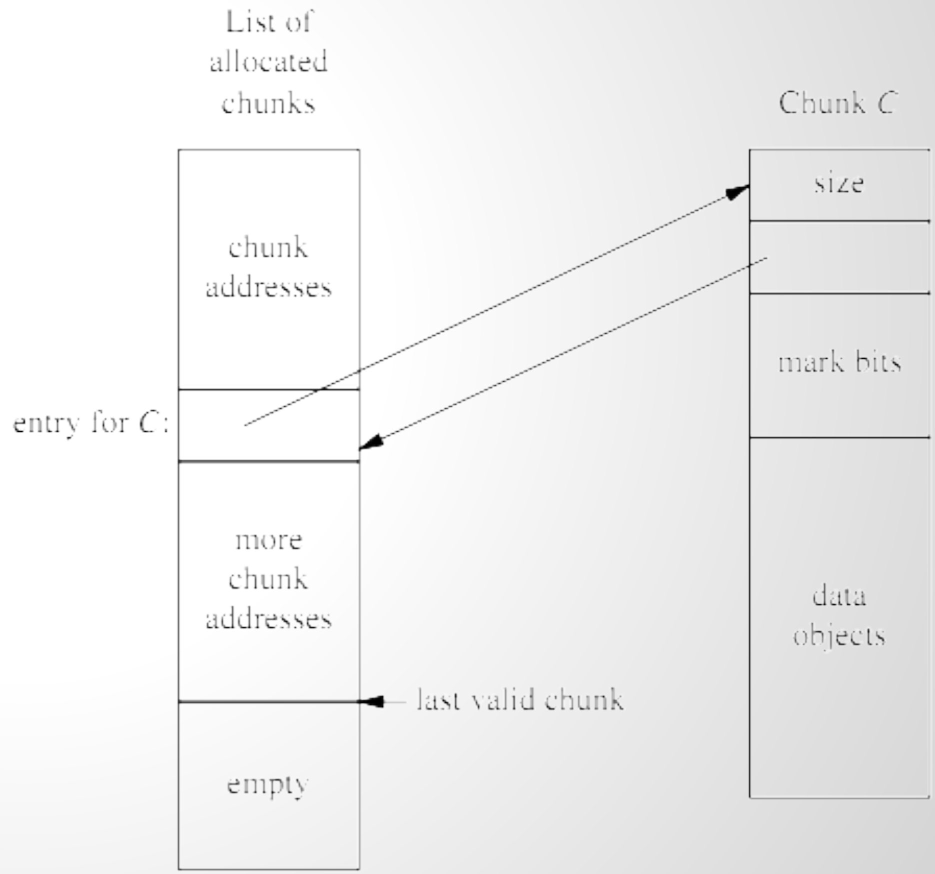
- A conservative GC for C/C++
- Why special?
  - Not type safe
  - Uncooperative: no good way to tell pointer from plain data
  - Memory layout restriction
- <gc_cpp.h>
  - overloads operator `new` for POD (plain old data) and classes without destructors
  - class `gc` overrides `new` and `delete` for classes with destructors

Boehm Garbage Collection (cont.)

- Metadata
  - Boehm GC stores objects in special memory “chunks”
  - Chunks store metadata in their headers
  - Objects are metadata-free
  - GC also maintains a list of allocated chunks
  - All chunks are aligned in memory

Boehm Garbage Collection (cont.)

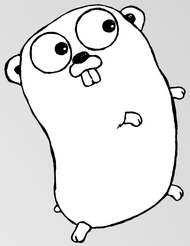
- Simple heuristics to identify pointers
  - Rule out: integers greater than the largest heap memory address and smaller than the smallest one
  - Metadata contains pointer to the entry in the chunk list
  - Use size info to check if pointer is valid



Boehm Garbage Collection (cont.)

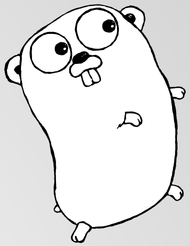
- Not perfect (duh!)
  - Likely to leak memory
  - Cannot handle fragmentation
- Acceptable overhead
  - Still has marking and sweeping phases
- Can be used as a leak detector

Go Lang 1.0



- mark-and-sweep (parallel implementation)
- non-generational
- non-compacting
- mostly precise
- stop-the-world
- bitmap-based representation
- zero-cost when the program is not allocating memory (that is: shuffling pointers around is as fast as in C, although in practice this runs somewhat slower than C because the Go compiler is not as advanced as C compilers such as GCC)
- supports finalizers on objects
- there is no support for weak references

Go Lang 1.4 (expected)



- hybrid stop-the-world/concurrent collector
- stop-the-world part limited by a 10ms deadline
- CPU cores dedicated to running the concurrent collector
- tri-color mark-and-sweep algorithm
- non-generational
- non-compacting
- fully precise
- incurs a small cost if the program is moving pointers around
- lower latency, but most likely also lower throughput, than Go 1.3 GC

OCaml

- Functional programming style involve large amount of small allocation
  - Generational GC
- *Minor heap:* small, fixed-size
- *Major heap*: larger, variable-size
- Heap compaction cycles

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
   Compilers: Principles, Techniques, and Tools, Second Edition
   Pearson Addison-Wesley, 2007, ISBN 0-321-48681-1
2. Hickey, Yaron Minsky. Anil Madhavapeddy. Jason.
   Real World OCaml; O'Reilly Media, Inc., 2013.
3. Brian Goetz
   Java theory and practice: A brief history of garbage collection http://www.ibm.com/developerworks/library/j-jtp10283/index.html
4. Jez Ng
   How the Boehm Garbage Collector Works
   http://discontinuously.com/2012/02/How-the-Boehm-Garbage-Collector-Works/
5. Vijay Saraswat
   Java is not type-safe
   http://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html
6. An Introduction to Garbage Collection by Richard Gillam
   http://icu-project.org/docs/papers/cpp_report/an_introduction_to_garbage_collection_part_i.html