

CS421 Lecture 20: Garbage Collection¹

Mark Hills
mhills@cs.uiuc.edu

University of Illinois at Urbana-Champaign

July 24, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

- 1 Motivation
- 2 Terminology
- 3 Reference Counting
- 4 Mark/Sweep
- 5 Copy Collectors (Scavengers)
- 6 Extensions

Static Allocation

- All memory is allocated up front for an entire program run
- Very fast: all memory addresses can be fixed at compile time
- Limited: does not allow recursion, allocation of data structures with a size dependent on run-time data

Stack Allocation

- Memory is allocated when procedures are called; lifetime is the lifetime of the stack frame
- Flexible: can now support recursion, local data structures with size dependent on parameter values
- Slightly dynamic: stack frames can outlive caller, provide limited version of "higher-order" functions
- Limited: procedure results limited to memory objects of a size known at compile time (since we need to know how much memory to use for it)

Heap Allocation

- Memory is allocated dynamically; lifetime dependent on usage of memory
- Flexible: allows for creation of memory objects with dynamic sizes, recursive structures (lists, trees, etc)
- Can return results of non-fixed size from procedures (in general, result has "fixed" size as size of reference or pointer)
- More natural model for higher-order functions (closures) – can live for arbitrary amount of time
- Downside: allocation and disposal more involved, requires explicit memory management or garbage collection

Garbage Collection

What is garbage collection?

Outline
Motivation
 Terminology
 Reference Counting
 Mark/Sweep
 Copy Collectors (Scavengers)
 Extensions

Memory Allocation
Why GC?

Garbage Collection

What is garbage collection?

Garbage collection is "the automatic management of dynamically allocated storage" (from Jones, *Garbage Collection*)

Mark Hills CS421 Lecture 20: Garbage Collection 6 / 27

Outline
Motivation
 Terminology
 Reference Counting
 Mark/Sweep
 Copy Collectors (Scavengers)
 Extensions

Memory Allocation
Why GC?

Why not Explicit Memory Management

- Advantage: high performance
- Disadvantage: double frees, memory leaks, dangling references, plus very challenging with some languages

Mark Hills CS421 Lecture 20: Garbage Collection 7 / 27

Outline
Motivation
 Terminology
 Reference Counting
 Mark/Sweep
 Copy Collectors (Scavengers)
 Extensions

Memory Allocation
Why GC?

Double Frees

```

1 Node *n = new Node();
2 ...
3 delete n;
4 ...
5 delete n;

```

- Accidental freeing of already freed memory common problem with explicit memory management
- In C++, corrupts memory subsystem, triggering later errors – hard to track down problem
- Challenge in libraries – who allocates storage? who deallocates? Languages contain no notation to help find bugs in usage...

Mark Hills CS421 Lecture 20: Garbage Collection 8 / 27

Outline
Motivation
 Terminology
 Reference Counting
 Mark/Sweep
 Copy Collectors (Scavengers)
 Extensions

Memory Allocation
Why GC?

Memory Leaks

- Memory object should be accessible as long as something points to it
- Memory not freed before last pointer to it is removed "leaks"
- Common source of out of memory errors

Mark Hills CS421 Lecture 20: Garbage Collection 9 / 27

Outline
Motivation
 Terminology
 Reference Counting
 Mark/Sweep
 Copy Collectors (Scavengers)
 Extensions

Memory Allocation
Why GC?

Dangling References

- Aliasing means that multiple variables could hold a reference to the same memory object
- Free'ing one variable, without cleaning up others, means that reference to freed memory still exists
- These references are "dangling references"; attempts to access memory through these can be disastrous

Mark Hills CS421 Lecture 20: Garbage Collection 10 / 27

Outline
Motivation
 Terminology
 Reference Counting
 Mark/Sweep
 Copy Collectors (Scavengers)
 Extensions

Memory Allocation
Why GC?

When to Free?

```

1 let wierdfun n =
2   let x = 3 and y = 4
3   in if n > 10 then (fun a -> a + x) else (fun b -> b + y)

```

- What memory is still accessible on return on wierdfun?
- Is there any way to know at compile time?
- Is there any way a programmer could properly clean memory up manually?

Mark Hills CS421 Lecture 20: Garbage Collection 11 / 27

Purpose of Garbage Collection

- Garbage collection solves these problems by removing responsibility of deleting allocated storage from user
- User generally still explicitly allocates new storage
- Language runtime recovers used storage when certain conditions are met, based on features of collector

Roots

- Roots provide a limited set of pointers into dynamically allocated memory
- Idea: all reachable memory should be reachable starting at one of the roots
- Example roots: global variables, object fields, function parameters on the stack

Garbage

- *Live* objects are those reachable from a root; provides a conservative estimate (not all live objects will be subsequently used)
- *Free* objects are those that have been returned to the memory management subsystem, either manually or by a collector
- *Garbage* objects are those that are not live and not free

Mutator and Collector

- Garbage collector sees layout of memory as primary data structure
- Running program changes (i.e. mutates) this memory layout – running program referred to as the *mutator*

Reference Counting: The Basics

- Memory objects extended with a counter
- Memory operations increment/decrement counter, based on type of operation (e.g., assigning a pointer to an object increments the counter of the object assigned to)
- Objects are garbage when counter goes to 0; can be immediately collected

Advantages

- Garbage collection cost amortized over entire program
- Collection can occur immediately
- No need to keep track of roots
- Good locality of reference (except, potentially, on cascades)

Disadvantages

- Pay a “tax” on each memory operation
- Reference counter has to know details about language’s memory operations, tied very closely to language semantics
- Incrementing counter takes CPU cycles
- Cyclic structures pose challenges

Mark/Sweep Collection: The Basics

- Collection starts with a predefined set of roots (mentioned earlier)
- Live objects reached by “tracing” pointer paths, setting a mark bit on memory objects
- Once all objects traced, unmarked objects discarded (sweep)

Advantages

- Can easily deal with cyclic structures
- Little overhead – no need to keep a counter
- No overhead on pointer operations, not as tightly coupled to the language

Disadvantages

- Can be slow: requires mutator to stop during GC, can cause substantial pauses
- Pauses make unusable for some systems (interactive games, hard real-time systems, etc)
- Can disrupt caching
- Can fragment memory
- If memory is tight, collector can thrash

Copy Collection: The Basics

- Memory space divided into two *semi-spaces*: *Fromspace* and *Tospace*
- At collection, *Fromspace* and *Tospace* flipped
- Collector traverses memory from roots, moving each live object to the other space
- Unmoved objects collected by being left behind

Advantages

- Same as mark/sweep
- Plus, takes care of fragmentation, can improve locality
- Easy to find next memory chunk to allocate – just look at current top of memory in *Tospace*
- Easy to tell when out of memory

Disadvantages

- Similar to mark/sweep: requires pausing mutator, may not be possible to use in some applications, can disrupt caching
- Also, requires more memory – need enough to make both semi-spaces

A Quick Note

GC methods presented so far have been presented in very generic, inefficient forms; research into GC methods have improved all of these, focusing on improving advantages and lessening disadvantages.

Combining Reference Counting with Mark/Sweep

- Reference counting has good performance characteristics, but can't deal with cyclic structures
- Idea: combine reference counting and mark/sweep
- One benefit: a smaller counter can be kept, mark/sweep can then set an appropriate counter value
- Also, can deal with cyclic structures, like standard mark/sweep algorithm

Generational Collectors

- One research question: can collectors be improved by taking application characteristics into account?
- OO languages: many objects are short lived; objects that live beyond a certain time tend to stay around a very long time
- Inefficient to try to collect long lived objects, which rarely die

Generational Collectors

- One research question: can collectors be improved by taking application characteristics into account?
- OO languages: many objects are short lived; objects that live beyond a certain time tend to stay around a very long time
- Inefficient to try to collect long lived objects, which rarely die
- Idea: group objects into generations, use different GC policies for each