Dynamic Memory Management

The runtime system linked in with the generated code should contain routines for allocation and deallocation of dynamic memory.

Pascal, C, C++, Modula-2 – Allocation and deallocation are *explicit*; i.e., the programmer is responsible to keep track of all allocated memory and when it is safe to free it.

Eiffel, Java – Allocation is explicit, but deallocation is *implicit*. Dynamic memory which is no longer used, is recycled by the *garbage collector*.

Ada – Implicit or explicit deallocation (implementation defined).

Modula-3 – Implicit and explicit deallocation (programmer's choice).

Lisp, Scheme, ML, Erlang, Haskell, Prolog, Mercury – Both allocation and deallocation of memory is *implicit*.

Memory Management

- In a language such as C or Pascal, there are three ways to allocate memory:

  1. Static allocation. Space for global variables is allocated at compile time.
  2. Stack allocation. Used for activation records (procedure call chains and local variables).
  3. Heap allocation. Used for data structures which are dynamically allocated either explicitly by the programmer (`new`, `malloc`, etc.) or implicitly (e.g., cons).

- The compiler and runtime system divide the available address space into three sections (one for each allocation type).

Memory Management (cont.)

- The static section is generated by the compiler and cannot be extended at run time.
- The stack grows and shrinks during execution, according to the depth of the call chain. Deep recursion often leads to stack overflow. A large number of parameters or locals can also result in running out of stack space.
- The heap grows by a suitable amount of memory when the program makes a request for more dynamic memory (e.g., by calling `malloc`).

Memory Organization

Program Code
Constant Static Data (strings, reals, . . .)
Uninitialized Static Data (Global variables)
Stack
$\Downarrow$ $\Uparrow$
Heap

Explicit Deallocation

Pascal's `new/dispose`, Modula-2's `ALLOCATE/DEALLOCATE`,
C's `malloc/free`, C++'s `new/delete`,
Ada's `new/unchecked_deallocation` (some implementations).

Problems:

- Dangling references

```
NEW(p); q := p; DISPOSE(p); use(q);
```

- Memory leaks
- Heap fragmentation

Implicit Deallocation

Lisp, Prolog – Equal-sized cells; not many changes to old cells

Eiffel, Modula-3, Java – Different-sized objects; frequent changes to old

- When to deallocate data?

  Disruptive – Stop execution of the program to perform GC whenever we run out of space.

  Real-Time/Incremental – Perform a partial GC for each pointer assigment or `new`.

  Concurrent – Run the GC as a separate thread/process.

- Fragmentation – Compact the heap as part of GC, or only when the GC fails to return a large enough block.

- Algorithms: Reference counting, mark-and-sweep, copying, generational.

Garbage Collection

- Heap-allocated records that are not reachable by any chain of pointers from program variables are *garbage*.
- Memory occupied by garbage should be reclaimed for use in allocating new records. This process is called garbage collection.
- GC is performed by the runtime system, not by the compiler; however, *GC usually requires compiler support*.
- GC frees programmers from the tedious and error-prone task of memory management, thus making a programming language with built-in GC higher-level.
- GC is a long-studied topic (many algorithms already developed).

```
let
  type list = {link: list,
               key: int}
  type tree = {key: int,
               left: tree,
               right: tree}
  function maketree() = ···
  function showtree(t: tree) = ···
 in
  let var x := list{link=nil,key=7}
      var y := list{link=x,key=9}
   in x.link := y
  end;
  let var p := maketree()
      var r := p.right
      var q := r.key
   in  garbage-collect here
      showtree(r)
  end
end
```

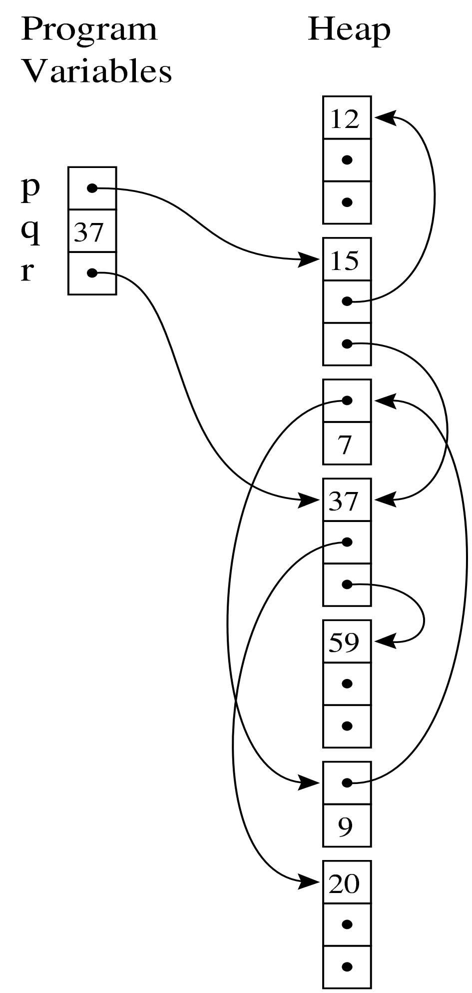


FIGURE 13.1. A heap to be garbage collected.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Finding the Object Graph

Program variables and dynamically allocated objects form a directed graph. The *roots* of this graph can be in:

1. global variables;
2. registers;
3. local variables & formal parameters on the stack.

Objects are *reachable* if there is a path of edges that leads to them starting from some root. Hence, the compiler must communicate to the garbage collector which registers/variables contain roots.

Mark-and-Sweep Garbage Collection

We can *mark* all reachable nodes using e.g. depth-first search.

```
function DFS(x)
  if x is a pointer into the heap
    if record x is not marked
      mark x
      for each field f_i of record x
          DFS(x.f_i)
```

ALGORITHM 13.2. Depth-first search.

From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Mark-and-Sweep Garbage Collection

- Any node not marked is garbage and is reclaimed by a *sweep* phase which scans the heap from beginning to end collecting all garbage cell in a *freelist* (a linked list). The sweep phase also unmarks all records.
- After garbage collection, the program resumes execution. New records are allocated from the freelist. When this is exhausted, it is a good time to invoke another garbage collection.

Mark-and-Sweep Garbage Collection (code)

Mark phase:

for each root v
    DFS(v)

Sweep phase:

$p \leftarrow$ first address in heap
while $p <$ last address in heap
    if record p is marked
        unmark p
    else let f_1 be the first field in p
        $p.f_1 \leftarrow$ `freelist`
        `freelist` $\leftarrow p$
    $p \leftarrow p$+(size of record p)

ALGORITHM 13.3. Mark-and-sweep garbage collection.
From *Modern Compiler Implementation in ML*,

Mark-and-Sweep Garbage Collection (example)

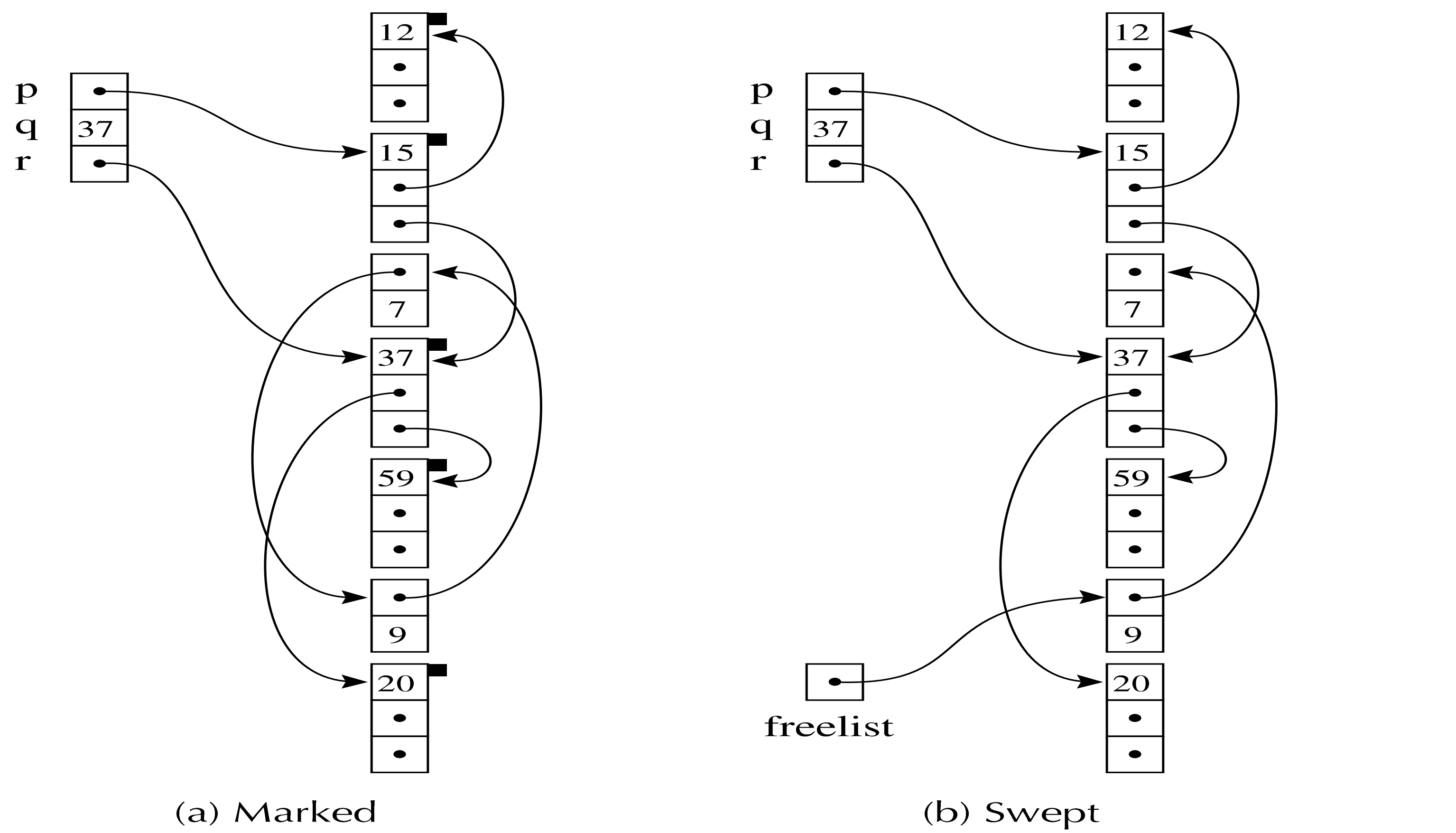


FIGURE 13.4. Mark-and-sweep collection.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Cost of Mark-and-Sweep Garbage Collection

Marking phase: takes time proportional to the amount of reachable data R.

Sweep phase: takes time proportional to the size of the heap H.

- The "good" that GC does is to recover $H - R$ words of usable memory. Thus, the *amortized cost* of GC (in instructions per allocated word) is:

$$\frac{c_1 R + c_2 H}{H - R}$$

- If $R \approx H$, the cost is very large. If $R \ll H$, the cost is $\approx c_2$.
- Usually, if $R/H > 50\%$, the collector increases H. Then the cost per allocated word will be approximately $c_1 + 2c_2$.

Tuning of Mark-and-Sweep Garbage Collection

Use of an explicit stack allows the marking phase to require a considerably smaller auxiliary area (H words rather than H activation records).

Pointer reversal allows use of the fields of the already processed heap records as auxiliary area (to store elements of the stack).

Use of an array of freelists allows variable-size allocation from the reclaimed space without searching ($freelist[i]$ stores all reclaimed records of size i).

Depth-first search using an explicit stack

```
function DFS(x)
 if x is a pointer and record x is not marked
  t ← 1
  stack[t] ← x
  while t > 0
        x ← stack[t];   t ← t − 1
        for each field f_i of record x
            if x.f_i is a pointer and record x.f_i is not marked
               mark x.f_i
               t ← t + 1;   stack[t] ← x.f_i
```

ALGORITHM 13.5. Depth-first search using an explicit stack.
From *Modern Compiler Implementation in ML,*

Depth-first search using pointer reversal

```
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 < # of fields in record x
          y ← x.f_i
          if y is a pointer and record y is 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
```

ALGORITHM 13.6. Depth-first search using pointer reversal.
From *Modern Compiler Implementation in ML*,

Reference Counting

- Identifying unreachable records can be done directly by the compiler maintaining a *reference count* field for records: a field that keeps track of how many pointers point to each record.

- Although reference counting seems simple and attractive, it also has severe problems:

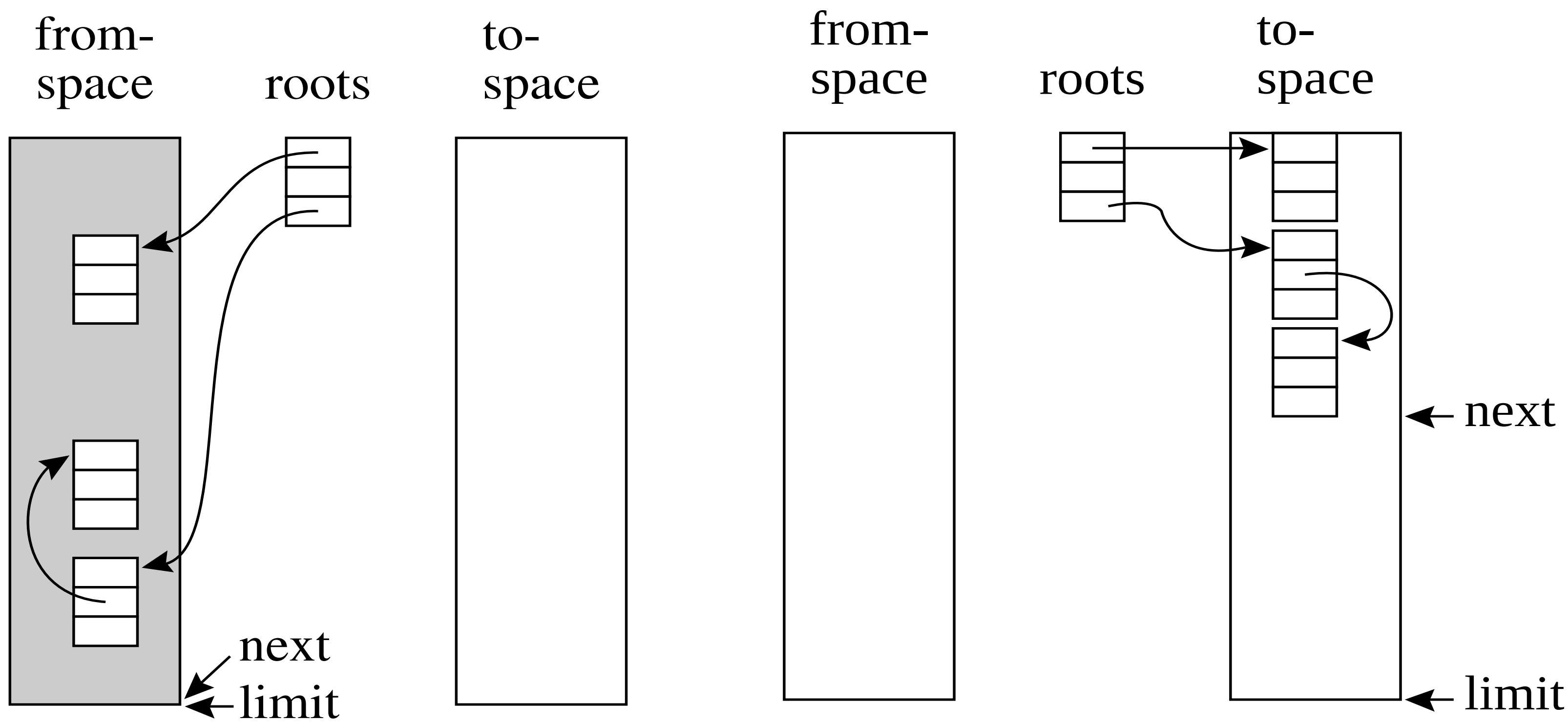
  1. Cycles of unreachable data cannot be detected and reclaimed.
  2. Keeping track of reference counts is very expensive.

- Because of these disadvantages, reference counting is *rarely used* for automatic storage management in programming language environments.

Copying Garbage Collection

- The allocated space is split in two *semi-spaces* and the garbage collector *copies* the reachable data from the old part of the heap (*from-space*) to the new part (*to-space*). Roots are made to point to the to-space and the roles of the two semi-spaces are swapped.
- After a copying GC, the to-space is *compact*: garbage is not interspersed with the reachable data.

Copying Garbage Collection



(a) Before collection (b) After collection

FIGURE 13.7. Copying collection.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Forwarding Pointers

Given a pointer p that points to from-space, make p point to to-space:

1. If p points to a from-space record that has already been copied, then $p.f_1$ is a special *forwarding pointer* that indicates where the copy is (so that other references to the record can be updated).
2. If p points to a from-space record that has not yet been copied, then it is copied to location `next` and the forwarding pointer is installed into $p.f_1$. This is safe as $p.f_1$ is not needed anymore.
3. If p is not a pointer at all, or p points outside from-space, then forwarding does nothing.

Algorithm for Pointer Forwarding

```
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
```

ALGORITHM 13.8. Forwarding a pointer.
From *Modern Compiler Implementation in ML,*

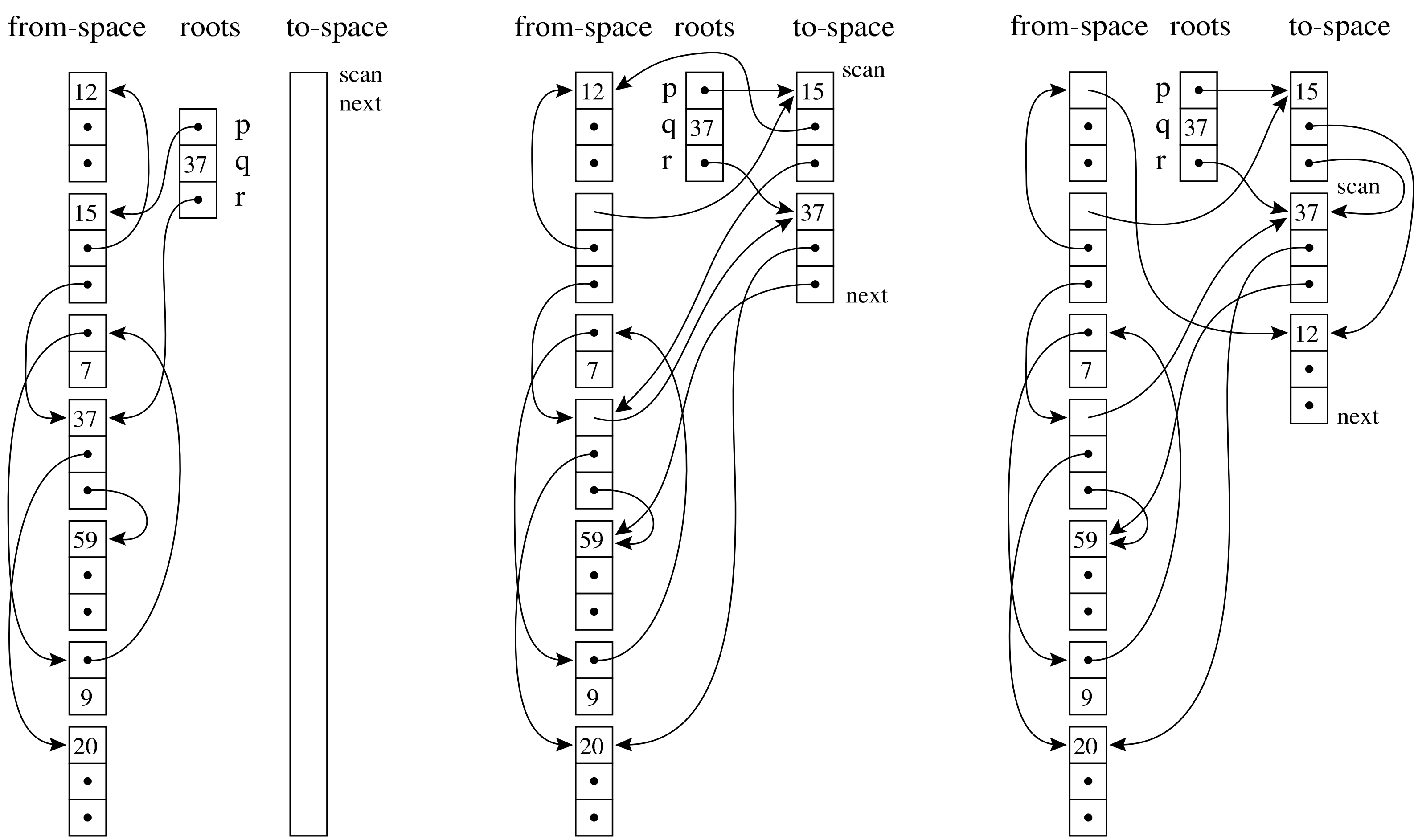
Cheney's Copying Garbage Collector

- Uses breadth-first search to traverse the reachable data.
- The algorithm is non-recursive, requires no external area nor pointer reversal, and is very simple to implement.
  - The roots are first forwarded.
  - The area between `scan` and `next` is used as a queue of all records that have been copied but whose fields have not yet been forwarded.
  - The area between the beginning of the to-space and `scan` contain records that have been copied and forwarded; all pointers to this area point to to-space.
- However, breadth-first search does not give as good locality of reference as depth-first search.

Cheney's Copying Garbage Collection Algorithm

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

ALGORITHM 13.9. Breadth-first copying garbage collection.
From *Modern Compiler Implementation in ML,*



(a) Before collection

(b) Roots forwarded

(c) One record scanned

FIGURE 13.10. Breadth-first copying collection.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Cost of Copying Garbage Collection

- GC takes time proportional to the amount of reachable data R.
- Each GC recovers $H/2 - R$ words of usable memory. Thus, the *amortized cost* of each collection (in instructions per allocated word) is:

$$\frac{c_3 R}{\frac{H}{2} - R}$$

- As $H \gg R$, the cost approaches zero; suggesting that there is *no inherent lower bound to the cost of garbage collection*.
- In a realistic setting, $H = 4R$, so GC cost is c_3 instructions per allocated word.

Incremental & Generational Garbage Collection

Incremental GC techniques allow memory reclamation to proceed piecemeal while applications are running. Incremental techniques can *reduce the disruptiveness* of garbage collection, and may even provide *real-time* guarantees. They can also be generalized into *concurrent collections* which proceed on another processor in parallel with actual program execution.

Generational GC schemes improve efficiency and/or locality by garbage collecting a smaller area more often, while *exploiting typical lifetime characteristics* to avoid undue overhead from long-lived objects. Because most collections are of a small area, typical pause times are also short, and for many applications this is an acceptable alternative to incremental collection.

Generational Garbage Collection

Newly created objects tend to die soon but objects that are reachable after many collections will probably survive for many collections more!

- Usually between 80–98% of all objects die within a few million instructions (or before a Mbyte has been allocated); the majority dies even more quickly (within tens of Kbytes of allocation).
- Long-lived objects are saved repeatedly by a simple copying collector.
- **Generational collection** avoids this repeated copying by segregating objects into multiple areas according to their age, and collecting younger areas more frequently than the older ones. Once objects have survived a number of GCs, they are moved into an older generation (*tenured*).

Generational Garbage Collection

- For generational GC to work, we must be able to collect younger generation(s) without having to examine the older one(s).
- In generational GC, roots are not just program variables; they also include any pointer from objects in old generations that points into objects in the generation being collected.
- These inter-generational references must be *remembered*: the compiler has to ensure that all `store` instructions check whether (pointer) updates to old objects cross the *write barrier*. This is done by remembered lists/sets, card/page marking using dirty bits.
- Fortunately, pointers from old to new objects are usually rare.

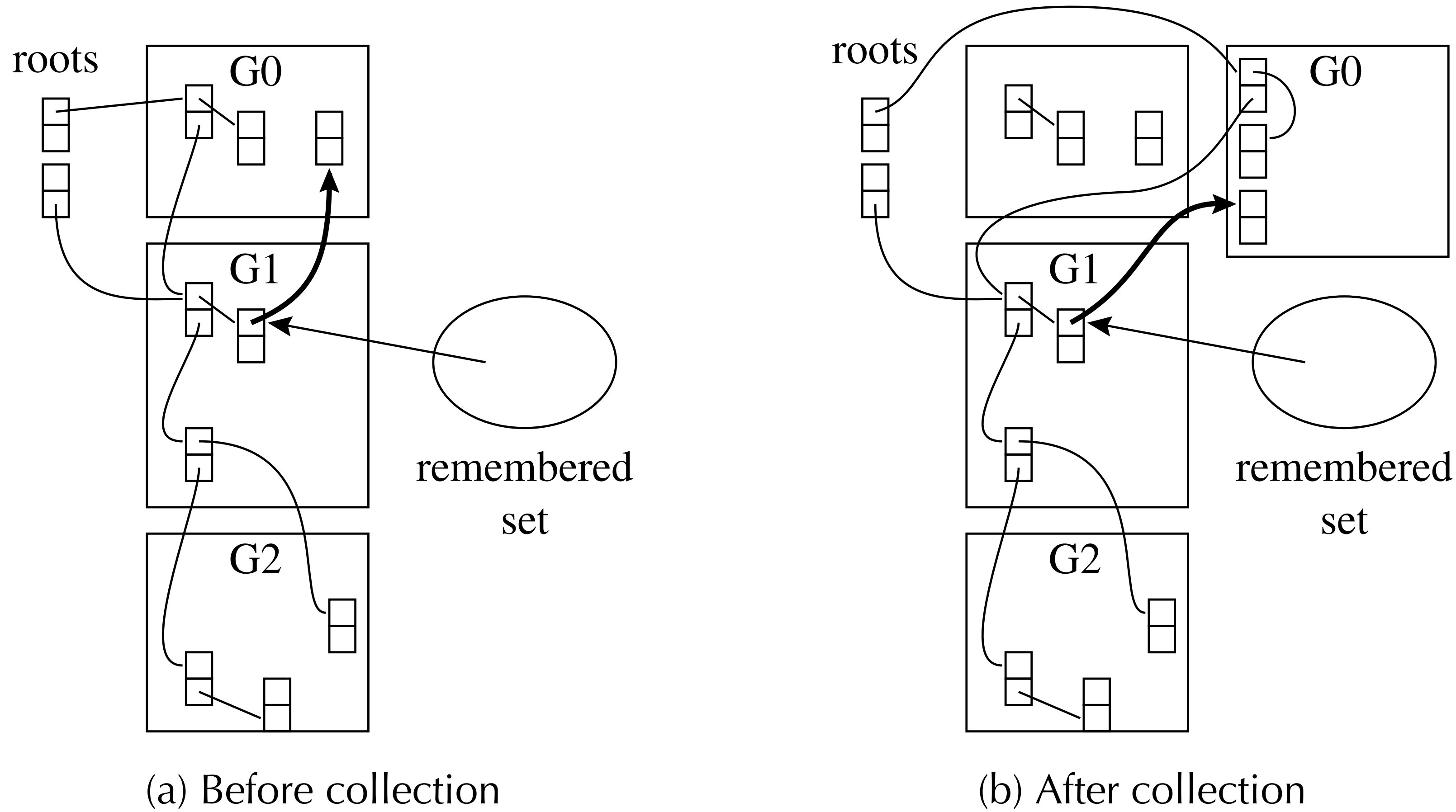


FIGURE 13.12. Generational collection. The bold arrow is one of the rare pointers from an older generation to a newer one.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Cost of Generational Garbage Collection

- In practice, it is common for the youngest generation to be less than 10% live data.
- With a copying collector, $H/R = 10$ *in this generation*, so the *amortized cost* per word reclaimed in each *minor collection* is:

$$\frac{c_3 R}{10R - R}$$

which is *very low* (about 1 instruction).

- Performing a *major collection* can be more expensive. Typically such collections are postponed as long as possible.
- Maintaining the remembered set also takes time; if the program updates many pointers from old to new objects, generational GC can be more expensive than non-generational GC!

Characteristics of Generational GC

- In practice, generational GC performs quite well: the majority of objects (that dies quickly) frees up enough space "free of cost" and copying the few ones that survive does not cost much.
- For *stop-and-collect* garbage collection, generational GC has the additional advantage of reducing the frequency of *disruptive pauses*. For many programs without hard real-time deadlines, this is sufficient for acceptable interactive use: most pauses are so brief that are not noticed by users.
- Generational techniques are often used as an acceptable substitute for more expensive incremental techniques.

Incremental Garbage Collection

- Incremental (and concurrent) techniques diminish long interruptions by interleaving GC work with program execution.

  > The *collector* tries to collect the garbage; meanwhile, the compiled program, called the *mutator*, keeps changing the graph of reachable data.

  Effectively, the GC work is spread out into more uniformly distributed parcels of smaller and (usually) bounded size.

- An *incremental* GC algorithm is one in which the collector operates only when the mutator requests.

- A *concurrent* GC algorithm is one which operates between or during any instructions executed by the mutator.

Incremental GC using Tricolor Marking

Classes of objects:

White objects are not yet visited by the depth-first or breadth-first search.

Grey objects have been visited (marked or copied), but their children have not yet been examined (e.g. in Cheney's algorithm these objects are between `scan` and `next`).

Black objects have been marked, and their children are also marked (e.g. in Cheney's algorithm these objects have been `scan`ned).

Starting with all objects white, objects pointed by roots are greyed. When there are no grey objects, then all the white objects are garbage.

Basic Tricolor Marking Algorithm

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

ALGORITHM 13.13. Basic tricolor marking.

From *Modern Compiler Implementation in ML,*

Preserving Tricolor Invariants

1. No black object points to a white object.
2. Every grey object is on the collector's scheduling stack or queue.

Write-barrier algorithms check every `store` by the mutator for preservation of 1&2 (e.g. whenever the mutator stores a white pointer a into a black object b, it colors b grey).

Read-barrier algorithms check all `fetch` instructions (e.g. whenever the mutator fetches a pointer b from any virtual memory page containing any non-black object, a page fault handler colors every object on the page black and makes all children of these objects grey).

Baker's Incremental Algorithm

Based on Cheney's copying algorithm; employs a read-barrier.

- Garbage collection starts with a *flip*: the roles of the from-space and to-space are swapped, and then all the roots are forwarded.
- Each time the mutator allocates new records, a few pointers of `scan` are processed, advancing `scan` towards `next` *by at least one word*. New records are allocated *at the end of the to-space* by decrementing `limit`.
- If a pointer fetched by the mutator points to from-space, the pointer is *forwarded* immediately. Thus, the mutator always has pointers only to the to-space; never to the from-space.

If the heap is divided into two semi-spaces of size $H/2$, and $R < H/4$, then `scan` will catch up with `next` *before* `next` reaches half-way through the to-space. At this point, no more than half the to-space will be occupied by newly allocated records.

Performance of Incremental GC

- Any implementation of write or read-barrier must synchronize with the collector. Software-based synchronization is expensive; in hardware, one can take advantage of the synchronization which is implicit in page faults: the OS ensures that no objects can access a faulting page before the fault is processed.
- In general, because incremental GC requires extra coordination between the mutator and the collector and higher conservatism, it is more expensive than blocking GC (where all the objects are reclaimed every time the collector is run).

Interaction with the Compiler

The compiler for a garbage-collected language interacts with the garbage collector by:

- generating code which allocates records (and initiates GC);
- describing locations of roots and those of them that are pointers;
- describing the layout of data records on the heap (i.e. determine the number of fields and indicate which fields are pointers).

For some versions of incremental collectors, the compiler must also generate instructions to implement the read or the write barrier.

Describing Data Layouts

- In statically typed or object-oriented languages, the simplest way of identifying heap objects is to have the first word of each object point to a special type or class descriptor record.
- This descriptor is generated by the static type information calculated by the semantic analysis phase of the compiler. It is passed to the `alloc()` function which initiates the GC.
- The *pointer map* (set of live temporaries that contain pointers) is best keyed by return addresses: for each pointer that is live immediately after a function call, the pointer map identifies its register or frame location. To mark/forward all the roots, GC starts at the top of the stack and scans downward, frame by frame.

Conservative Garbage Collection

- The compiler does *not* inform the collector which variables and record-fields contain variables, so the collector must “guess”.
- Any bit-pattern pointing into the allocated heap is assumed to be a possible pointer and keeps its pointed-to record live. However, since the bit-pattern might really be an integer, the object cannot be moved and some garbage objects might not be reclaimed.
- Conservative GC might thus occasionally suffer from disastrous space leaks; several techniques for making these situations unlikely exist.