13 Garbage Collection

gar-bage: unwanted or useless material

Webster's Dictionary

Heap-allocated records that are not reachable by any chain of pointers from program variables are *garbage*. The memory occupied by garbage should be reclaimed for use in allocating new records. This process is called *garbage collection*, and is performed not by the compiler but by the runtime system (the support programs linked with the compiled code).

Ideally, we would say that any record that is not dynamically live (will not be used in the future of the computation) is garbage. But, as Section 10.1 explains, it is not always possible to know whether a variable is live. So we will use a conservative approximation: we will require the compiler to guarantee that any *live* record is *reachable*; we will ask the compiler to minimize the number of reachable records that are *not* live; and we will preserve all reachable records, even if some of them might not be live.

Figure 13.1 shows a Tiger program ready to undergo garbage collection (at the point marked *garbage-collect here*). There are only three program variables in scope: `p`, `q`, and `r`.

13.1 MARK-AND-SWEEP COLLECTION

Program variables and heap-allocated records form a directed graph. The variables are *roots* of this graph. A node n is reachable if there is a path of directed edges $r \to \cdots \to n$ starting at some root r. A graph-search algorithm such as *depth-first search* (Algorithm 13.2) can *mark* all the reachable nodes.

```
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
```

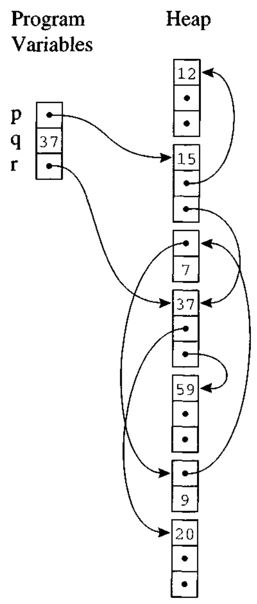


FIGURE 13.1. A heap to be garbage collected.

```
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.

Any node not marked must be garbage, and should be reclaimed. This can be done by a *sweep* of the entire heap, from its first address to its last, looking for nodes that are not marked (Algorithm 13.3). These are garbage and can be linked together in a linked list (the *freelist*). The sweep phase should also unmark all the marked nodes, in preparation for the next garbage collection.

After the garbage collection, the compiled program resumes execution. Whenever it wants to heap-allocate a new record, it gets a record from the freelist. When the freelist becomes empty, that is a good time to do another garbage collection to replenish the freelist.

```
Mark phase:                    Sweep phase:
  for each root $v$              $p \leftarrow$ first address in heap
      DFS($v$)                   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.

Cost of garbage collection. Depth-first search takes time proportional to the number of nodes it marks, that is, time proportional to the amount of reachable data. The sweep phase takes time proportional to the size of the heap. Suppose there are R words of reachable data in a heap of size H. Then the cost of one garbage collection is $c_1 R + c_2 H$ for some constants c_1 and c_2; for example, c_1 might be 10 instructions and c_2 might be 3 instructions.

The "good" that collection does is to replenish the freelist with $H - R$ words of usable memory. Therefore, we can compute the *amortized cost* of collection by dividing the *time spent collecting* by the *amount of garbage reclaimed*. That is, for every word that the compiled program allocates, there is an eventual garbage-collection cost of

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

If R is close to H, this cost becomes very large: each garbage collection reclaims only a few words of garbage. If H is much larger than R, then the cost per allocated word is approximately c_2, or about three instructions of garbage-collection cost per word allocated.

The garbage collector can measure H (the heap size) and $H - R$ (the freelist size) directly. After a collection, if R/H is larger than 0.5 (or some other criterion), the collector should increase H by asking the operating system for more memory. Then the cost per allocated word will be approximately $c_1 + 2c_2$, or perhaps 16 instructions per word.

Using an explicit stack. The DFS algorithm is recursive, and the maximum depth of its recursion is as long as the longest path in the graph of reachable

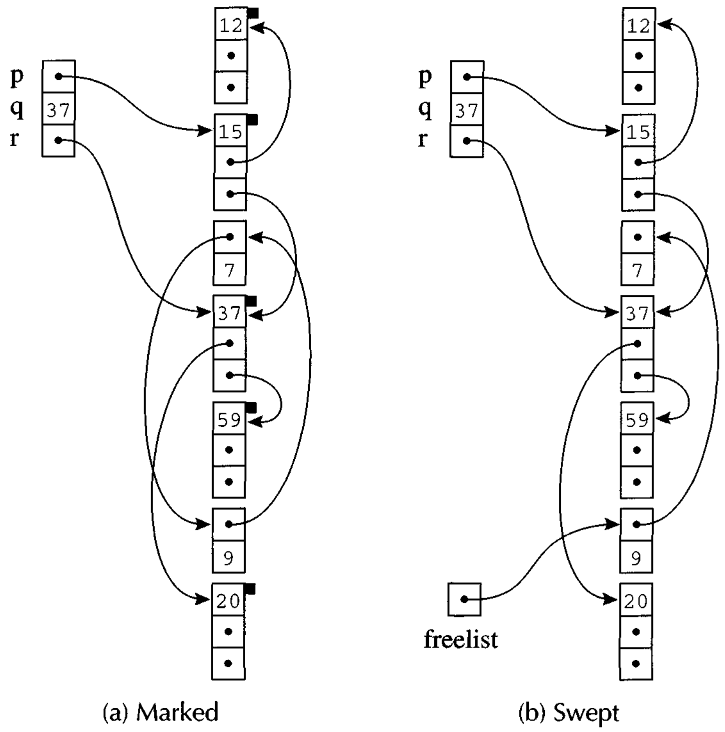


FIGURE 13.4. Mark-and-sweep collection.

data. There could be a path of length H in the worst case, meaning that the stack of activation records would be larger than the entire heap!

To attack this problem, we use an explicit stack (instead of recursion), as in Algorithm 13.5. Now the stack could still grow to size H, but at least this is H words and not H activation records. Still, it is unacceptable to require auxiliary stack memory as large as the heap being collected.

Pointer reversal. After the contents of field $x.f_i$ has been pushed on the stack, Algorithm 13.5 will never again look the original location $x.f_i$. This means we can use $x.f_i$ to store one element of the stack itself! This all-too-clever idea is called *pointer reversal*, because $x.f_i$ will be made to point back to the record from which x was reached. Then, as the stack is popped, the field $x.f_i$ will be restored to its original value.

Algorithm 13.6 requires a field in each record called *done*, which indicates how many fields in that record have been processed. This takes only a few

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

ALGORITHM 13.5. Depth-first search using an explicit stack.

bits per record (and it can also serve as the mark field).

The variable t serves as the top of the stack; every record x on the stack is already marked, and if i = done[x] then $x.f_i$ is the "stack link" to the next node down. When popping the stack, $x.f_i$ is restored to its original value.

An array of freelists. The sweep phase is the same no matter which marking algorithm is used: it just puts the unmarked records on the freelist, and unmarks the marked records. But if records are of many different sizes, a simple linked list will not be very efficient for the allocator. When allocating a record of size n, it may have to search a long way down the list for a free block of that size.

A good solution is to have an array of several freelists, so that freelist[i] is a linked list of all records of size i. The program can allocate a node of size i just by taking the head of freelist[i]; the sweep phase of the collector can put each node of size j at the head of freelist[j].

If the program attempts to allocate from an empty freelist[i], it can try to grab a larger record from freelist[j] (for $j > i$) and split it (putting the unused portion back on freelist[$j - i$]). If this fails, it is time to call the garbage collector to replenish the freelists.

Fragmentation. It can happen that the program wants to allocate a record of size n, and there are many free records smaller than n but none of the right size. This is called *external fragmentation*. On the other hand, *internal fragmentation* occurs when the program uses a too-large record without splitting it, so that the unused memory is inside the record instead of outside.

```
function DFS($x$)
 if $x$ is a pointer and record $x$ is not marked
  $t \leftarrow$ nil
  mark $x$;  done[$x$] $\leftarrow 0$
  while true
        $i \leftarrow$ done[$x$]
        if $i < \#$ of fields in record $x$
          $y \leftarrow x.f_i$
          if $y$ is a pointer and record $y$ is not marked
            $x.f_i \leftarrow t$;  $t \leftarrow x$;  $x \leftarrow y$
            mark $x$;  done[$x$] $\leftarrow 0$
          else
            done[$x$] $\leftarrow i + 1$
        else
          $y \leftarrow x$;  $x \leftarrow t$
          if $x$ = nil then return
          $i \leftarrow$ done[$x$]
          $t \leftarrow x.f_i$;  $x.f_i \leftarrow y$
          done[$x$] $\leftarrow i + 1$
```

ALGORITHM 13.6. Depth-first search using pointer reversal.

13.2 REFERENCE COUNTS

One day a student came to Moon and said: "I understand how to make a better garbage collector. We must keep a reference count of the pointers to each cons."
Moon patiently told the student the following story:
"One day a student came to Moon and said: 'I understand how to make a better garbage collector ...' "

(MIT-AI koan by Danny Hillis)

Mark-sweep collection identifies the garbage by first finding out what is reachable. Instead, it can be done directly by keeping track of how many pointers point to each record: this is the *reference count* of the record, and it is stored with each record.

The compiler emits extra instructions so that whenever p is stored into $x.f_i$, the reference count of p is incremented, and the reference count of what $x.f_i$ previously pointed to is decremented. If the decremented reference count of some record r reaches zero, then r is put on the freelist and all the other records that r points to have their reference counts decremented.

Instead of decrementing the counts of $r.f_i$ when r is put on the freelist, it is better to do this "recursive" decrementing when r is removed from the freelist, for two reasons:

1. It breaks up the "recursive decrementing" work into shorter pieces, so that the program can run more smoothly (this is important only for interactive or real-time programs).
2. The compiler must emit code (at each decrement) to check whether the count has reached zero and put the record on the freelist, but the recursive decrementing will be done only in one place, in the allocator.

Reference counting seems simple and attractive. But there are two major problems:

1. Cycles of garbage cannot be reclaimed. In Figure 13.1, for example, there is a loop of list cells (whose keys are 7 and 9) that are not reachable from program variables; but each has a reference count of 1.
2. Incrementing the reference counts is very expensive indeed. In place of the single machine instruction $x.f_i \leftarrow p$, the program must execute

```
$z$        $\leftarrow x.f_i$
$c$        $\leftarrow z.\text{count}$
$c$        $\leftarrow c - 1$
$z.\text{count}$  $\leftarrow c$
if $c = 0$ call $putOnFreelist$
$x.f_i$     $\leftarrow p$
$c$        $\leftarrow p.\text{count}$
$c$        $\leftarrow c + 1$
$p.\text{count}$  $\leftarrow c$
```

A naive reference counter will increment and decrement the counts on every assignment to a program variable. Because this would be extremely expensive, many of the increments and decrements are eliminated using dataflow analysis: As a pointer value is fetched and then propagated through local variables, the compiler can aggregate the many changes in the count to a single increment, or (if the net change is zero) no extra instructions at all. However, even with this technique there are many ref-count increments and decrements that remain, and their cost is very high.

There are two possible solutions to the "cycles" problem. The first is simply to require the programmer to explicitly break all cycles when she is done with a data structure. This is less annoying than putting explicit *free* calls (as would be necessary without any garbage collection at all), but it is hardly elegant. The other solution is to combine reference counting (for eager and nondisruptive reclamation of garbage) with an occasional mark-sweep collection (to reclaim the cycles).

On the whole, the problems with reference counting outweigh its advantages, and it is rarely used for automatic storage management in programming language environments.

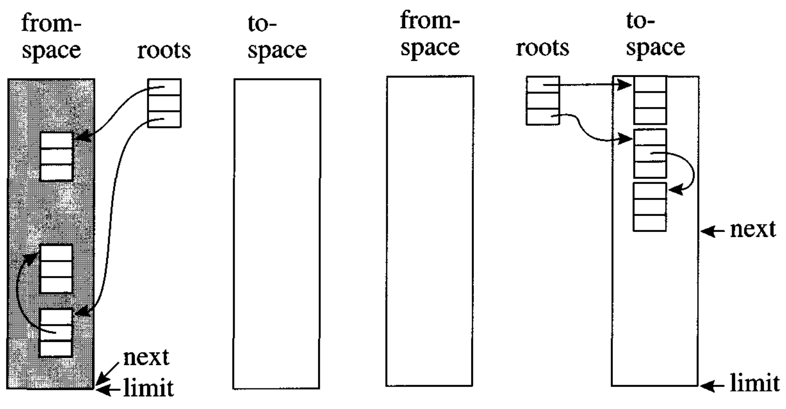
13.3 COPYING COLLECTION

The reachable part of the heap is a directed graph, with records as nodes, and pointers as edges, and program variables as roots. Copying garbage collection traverses this graph (in a part of the heap called *from-space*), building an isomorphic copy in a fresh area of the heap (called *to-space*). The to-space copy is *compact*, occupying contiguous memory without fragmentation (that is, without free records interspersed with the reachable data). The roots are made to point at the to-space copy; then the entire from-space (garbage, plus the previously reachable graph) is unreachable.

Figure 13.7 illustrates the situation before and after a copying collection. Before the collection, from-space is full of reachable nodes and garbage; there is no place left to allocate, since `next` has reached `limit`. After the collection, the area of to-space between `next` and `limit` is available for the compiled program to allocate new records. Because the new-allocation area is contiguous, allocating a new record of size n into pointer `p` is very easy: just copy `next` to `p`, and increment `next` by n. Copying collection does not have a fragmentation problem.

Eventually, the program will allocate enough that `next` reaches `limit`; then another garbage collection is needed. The roles of from-space and to-space are swapped, and the reachable data are again copied.

Initiating a collection. To start a new collection, the pointer `next` is initialized to point at the beginning of to-space; as each reachable record in from-space is found, it is copied to to-space at position `next`, and `next` incremented by the size of the record.



(a) Before collection (b) After collection

FIGURE 13.7. Copying collection.

```
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.$f_i$ $\leftarrow$ $p.f_i$
                $p.f_1$ $\leftarrow$ next
                next $\leftarrow$ next$+$ size of record $p$
                return $p.f_1$
    else return $p$
```

ALGORITHM 13.8. Forwarding a pointer.

Forwarding. The basic operation of copying collection is *forwarding* a pointer; that is, given a pointer `p` that points to from-space, make `p` point to to-space (Algorithm 13.8).

There are three cases:

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. The forwarding pointer can be identified just by the fact that it points within the to-space, as no ordinary from-space field could point there.
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 $p.f_1$. It's

```
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
```

ALGORITHM 13.9. Breadth-first copying garbage collection.

all right to overwrite the f_1 field of the old record, because all the data have already been copied to the to-space at next.

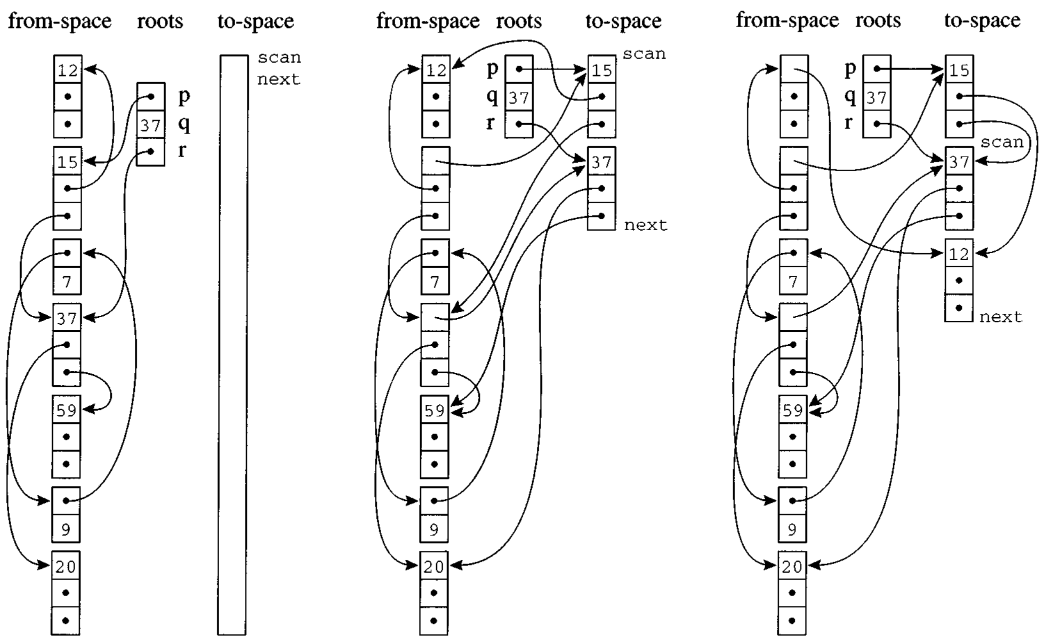
3. If p is not a pointer at all, or if it points outside from-space (to a record outside the garbage-collected arena, or to to-space), then forwarding p does nothing.

Cheney's algorithm. The simplest algorithm for copying collection uses breadth-first search to traverse the reachable data (Algorithm 13.9, illustrated in Figure 13.10). First, the roots are forwarded. This copies a few records (those reachable *directly* from root pointers) to to-space, thereby incrementing next.

The area between scan and next contains records that have been copied to to-space, but whose fields have not yet been forwarded: in general, these fields point to from-space. The area between the beginning of to-space and scan contains records that have been copied *and* forwarded, so that all the pointers in this area point to to-space. The **while** loop of (Algorithm 13.9) moves scan toward next, but copying records will cause next to move also. Eventually, scan catches up with next after all the reachable data are copied to to-space.

Cheney's algorithm requires no external stack, and no pointer reversal: it uses the to-space area between scan and next as the queue of its breadth-first search. This makes it considerably simpler to implement than depth-first search with pointer reversal.

Locality of reference. However, pointer data structures copied by breadth-first have poor locality of reference: If a record at address a points to another record at address b, it is likely that a and b will be far apart. Conversely, the record at $a+8$ is likely to be unrelated to the one at a. Records that are copied near each other are those whose distance from the roots are equal.



(a) Before collection (b) Roots forwarded (c) One record scanned

FIGURE 13.10. Breadth-first copying collection.

In a computer system with virtual memory, or with a memory cache, good locality of reference is important. After the program fetches address a, then the memory subsystem expects addresses near a to be fetched soon. So it ensures that the entire page or cache line containing a and nearby addresses can be quickly accessed.

Suppose the program is fetching down a chain of n pointers in a linked list. If the records in the list are scattered around memory, each on a page (or cache line) containing completely unrelated data, then we expect n difference pages or cache lines to be active. But if successive records in the chain are at adjacent addresses, then only n/k pages (cache lines) need to be active, where k records fit on each page (cache line).

Depth-first copying gives better locality, since each object a will tend to be adjacent to its first child b; unless b is adjacent to another "parent" a'. Other children of a may not be adjacent to a, but if the subtree b is small, then they should be nearby.

But depth-first copy requires pointer-reversal, which is inconvenient and

```
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
     $q \leftarrow$ next
     next $\leftarrow$ next+ size of record $p$
     $r \leftarrow$ nil
     for each field $f_i$ of record $p$
         $q.f_i \leftarrow p.f_i$
         if $q.f_i$ points to from-space and $q.f_i.f_1$ does not point to to-space
         then $r \leftarrow q.f_i$
     $p.f_1 \leftarrow q$
     $p \leftarrow r$
   until $p$ = nil
```

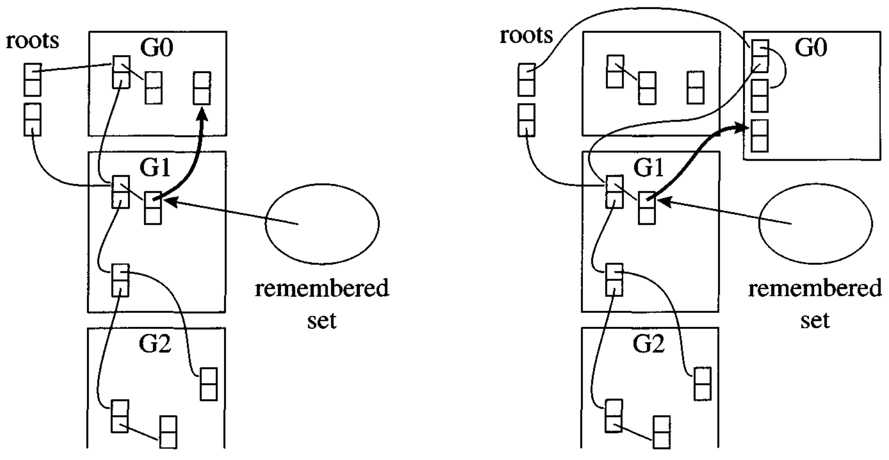
ALGORITHM 13.11. Semi-depth-first forwarding.

slow. A hybrid, partly depth-first and partly breadth-first algorithm can provide acceptable locality. The basic idea is to use breadth-first copying, but whenever an object is copied, see if some child can be copied near it (Algorithm 13.11).

Cost of garbage collection. Breadth-first search (or the semi-depth-first variant) takes time proportional to the number of nodes it marks; that is, c_3R for some constant c_3 (perhaps equal to 10 instructions). There is no sweep phase, so c_3R is the total cost of collection. The heap is divided into two semi-spaces, so each collection reclaims $H/2 - R$ words that can be allocated before the next collection. The amortized cost of collection is thus

$$\frac{c_3R}{\frac{H}{2} - R}$$

instructions per word allocated.



(a) Before collection (b) After collection

FIGURE 13.12. Generational collection. The bold arrow is one of the rare pointers from an older generation to a newer one.

As H grows much larger than R, this cost approaches zero. That is, *there is no inherent lower bound to the cost of garbage collection.* In a more realistic setting, where $H = 4R$, the cost would be about 10 instructions per word allocated. This is rather costly in space and time: it requires four times as much memory as reachable data, and requires 40 instructions of overhead for every 4-word object allocated. To reduce both space and time costs significantly, we use *generational* collection.

13.4 GENERATIONAL COLLECTION

In many programs, newly created objects are likely to die soon; but an object that is still reachable after many collections will probably survive for many collections more. Therefore the collector should concentrate its effort on the "young" data, where there is a higher proportion of garbage.

We divide the heap into *generations*, with the youngest objects in generation G_0; every object in generation G_1 is older than any object in G_0; everything in G_2 is older than anything in G_1 and so on.

To collect (by mark-and-sweep or by copying) just G_0, just start from the roots and do either depth-first marking or breadth-first copying (or semi-depth-first copying). But now the roots are not just program variables: they include any pointer within $G_1, G_2, \ldots$ that points into G_0. If there are too many of these, then processing the roots will take longer than the traversal of reachable objects within G_0!

Fortunately, it is rare for an older object to point to a much younger object. In many common programming styles, when an object a is created its fields are immediately initialized; for example, they might be made to point to b and c. But b and c already exist; they are older than a. So we have a newer object pointing to an older object. The only way that an older object b could point to a newer object a is if some field of b is updated long after b is created; this turns out to be rare.

To avoid searching all of $G_1, G_2, \ldots$ for roots of G_0, we make the compiled program *remember* where there are pointers from old objects to new ones. There are several ways of remembering:

Remembered list: The compiler generates code, after each *update* store of the form $b.f_i \leftarrow a$, to put b into a vector of *updated objects*. Then, at each garbage collection, the collector scans the remembered list looking for old objects b that point into G_0.

Remembered set: Like the remembered list, but uses a bit within object b to record that b is already in the vector. Then the code generated by the compiler can check this bit to avoid duplicate references to b in the vector.

Card marking: Divide memory into logical "cards" of size 2^k bytes. An object can occupy part of a card or can start in the middle of one card and continue onto the next. Whenever address b is updated, the card containing that address is *marked*. There is an array of bytes that serve as marks; the byte index can be found by shifting address b right by k bits.

Page marking: This is like card marking, but if 2^k is the page size, then the computer's virtual memory system can be used instead of extra instructions generated by the compiler. Updating an old generation sets a *dirty bit* for that page. If the operating system does not make dirty bits available to user programs, then the user program can implement this by write-protecting the page and asking the operating system to refer protection violations to a user-mode fault handler that records the dirtiness and unprotects the page.

When a garbage collection begins, the remembered set tells which objects (or cards, or pages) of the old generation can possibly contain pointers into G_0; these are scanned for roots.

Algorithm 13.3 or 13.9 can be used to collect G_0: "heap" or "from-space" means G_0, "to-space" means a new area big enough to hold the reachable objects in G_0, and "roots" include program variables *and* the remembered set. Pointers to older generations are left unchanged: the marking algorithm does not mark old-generation records, and the copying algorithm copies them verbatim without forwarding them.

After several collections of G_0, generation G_1 may have accumulated a significant amount of garbage that should be collected. Since G_0 may contain many pointers into G_1, it is best to collect G_0 and G_1 together. As before, the remembered set must be scanned for roots contained in $G_2, G_3, \ldots$. Even more rarely, G_2 will be collected, and so on.

Each older generation should be exponentially bigger than the previous one. If G_0 is half a megabyte, then G_1 should be two megabytes, G_2 should be eight megabytes, and so on. An object should be promoted from G_i to G_{i+1} when it survives two or three collections of G_i.

Cost of generational collection. Without detailed empirical information about the distribution of object lifetimes, we cannot analyze the behavior of generational collection. In practice, however, it is common for the youngest generation to be less than 10% live data. With a copying collector, this means that H/R is 10 *in this generation*, so that the amortized cost per word reclaimed is $c_3R/(10R - R)$, or about 1 instruction. If the amount of reachable data in G_0 is about 50 to 100 kilobytes, then the amount of space "wasted" by having $H = 10R$ in the youngest generation is about a megabyte. In a 50-megabyte multigeneration system, this is a small space cost.

Collecting the older generations can be more expensive. To avoid using too much space, a smaller H/R ratio can be used for older generations. This increases the time cost of an older-generation collection, but these are sufficiently rare that the overall amortized time cost is still good.

Maintaining the remembered set also takes time, approximately 10 instructions per pointer update to enter an object into the remembered set and then process that entry in the remembered set. If the program does many more updates than fresh allocations, then generational collection may be more expensive than nongenerational collection.

13.5 INCREMENTAL COLLECTION

Even if the overall garbage collection time is only a few percent of the computation time, the collector will occasionally interrupt the program for long periods. For interactive or real-time programs this is undesirable. Incremental or concurrent algorithms interleave garbage collection work with program execution to avoid long interruptions.

```
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.

Terminology. The *collector* tries to collect the garbage; meanwhile, the compiled program keeps changing (mutating) the graph of reachable data, so it is called the *mutator*. An *incremental* algorithm is one in which the collector operates only when the mutator requests it; in a *concurrent* algorithm the collector can operate between or during any instructions executed by the mutator.

Tricolor marking. In a mark-sweep or copying garbage collection, there are three classes of records:

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. In mark-sweep collection, these objects are on the stack; in Cheney's copying collection, they are between `scan` and `next`.

Black objects have been marked, and their children also marked. In mark-sweep collection, they have already been popped off the stack; in Cheney's algorithm, have already been scanned.

The collection starts with all objects white; the collector executes Algorithm 13.13, blackening grey objects and greying their white children. Implicit in changing an object from grey to black is *removing it from the stack or queue*; implicit in greying an object is *putting it into the stack or queue*. When there are no grey objects, then all white objects must be garbage.

Algorithm 13.13 generalizes all of the mark-sweep and copying algorithms shown so far: Algorithms 13.2, 13.3, 13.5, 13.6, and 13.9.

All these algorithms preserve two natural invariants:

1. No black object points to a white object.
2. Every grey object is on the collector's (stack or queue) data structure (which we will call the *grey-set*).

While the collector operates, the mutator creates new objects (of what color?) and updates pointer fields of existing objects. If the mutator breaks one of the invariants, then the collection algorithm will not work.

Most incremental and concurrent collection algorithms are based on techniques to allow the mutator to get work done while preserving the invariants. For example:

Dijkstra, Lamport, et al. Whenever the mutator stores a white pointer *a* into a black object *b*, it colors *a* grey. (The compiler generates extra instructions at each store to check for this.)

Steele. Whenever the mutator stores a white pointer *a* into a black object *b*, it colors *b* grey (using extra instructions generated by the compiler).

Boehm, Demers, Shenker. All-black pages are marked read-only in the virtual memory system. Whenever the mutator stores *any* value into an all-black page, a page fault marks all objects on that page grey (and makes the page writable).

Baker. Whenever the mutator fetches a pointer *b* to a white object, it colors *b* grey. The mutator never possesses a pointer to a white object, so it cannot violate invariant 1. The instructions to check the color of *b* are generated by the compiler after every fetch.

Appel, Ellis, Li. Whenever the mutator fetches a pointer *b* from any virtual-memory page containing any nonblack object, a page-fault handler colors every object on the page black (making children of these objects grey). Thus the mutator never possesses a pointer to a white object.

The first three of these are *write-barrier* algorithms, meaning that each *write* (store) by the mutator must be checked to make sure an invariant is preserved. The last two are *read-barrier* algorithms, meaning that *read* (fetch) instructions are the ones that must be checked. We have seen write barriers before, for generational collection: remembered lists, remembered sets, card marking, and page marking are all different implementations of the write barrier. Similarly, the read barrier can be implemented in software (as in Baker's algorithm) or using the virtual-memory hardware.

Any implementation of a write or read barrier must synchronize with the collector. For example, a Dijkstra-style collector might try to change a white node to grey (and put it into the grey-set) at the same time the mutator is also greying the node (and putting it into the grey-set). Thus, software implementations of the read or write barrier will need to use explicit synchronization instructions, which can be expensive.

But implementations using virtual-memory hardware can take advantage of the synchronization implicit in a page fault: if the mutator faults on a page, the operating system will ensure that no other process has access to that page before processing the fault.

13.6 BAKER'S ALGORITHM

Baker's algorithm illustrates the details of incremental collection. It is based on Cheney's copying collection algorithm, so it forwards reachable objects from from-space to to-space. Baker's algorithm is compatible with generational collection, so that the from-space and to-space might be for generation G_0, or might be $G_0 + \cdots + G_k$.

To initiate a garbage collection (which happens when an *allocate* request fails for lack of unused memory), the roles of the (previous) from-space and to-space are swapped, and all the roots are forwarded; this is called the *flip*. Then the mutator is resumed; but each time the mutator calls the allocator to get a new record, a few pointers at `scan` are scanned, so that `scan` advances toward `next`. Then a new record is allocated *at the end of the to-space* by decrementing `limit` by the appropriate amount.

The invariant is that the mutator has pointers only to to-space (never to from-space). Thus, when the mutator allocates and initializes a new record, that record need not be scanned; when the mutator stores a pointer into an old record, it is only storing a to-space pointer.

If the mutator fetches a field of a record, it might break the invariant. So each fetch is followed by two or three instructions that check whether the fetched pointer points to from-space. If so, that pointer must be *forwarded* immediately, using the standard *forward* algorithm.

For every word allocated, the allocator must advance `scan` by at least one word. When `scan=next`, the collection terminates until the next time the allocator runs out of 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 halfway through the to-space; also by this time, no more than half the to-space will be occupied by newly allocated records.

Baker's algorithm copies no more data than is live at the flip. Records allocated during collection are not scanned, so they do not add to the cost of collection. The collection cost is thus $c_3 R$. But there is also a cost to check (at every allocation) whether incremental scanning is necessary; this is proportional to $H/2 - R$.

But the largest cost of Baker's algorithm is the extra instructions after every fetch, required to maintain the invariant. If one in every 10 instructions fetches from a heap record, and each of these fetches requires two extra instructions to test whether it is a from-space pointer, then there is at least a

20% overhead cost just to maintain the invariant. All of the incremental or concurrent algorithms that use a software write or read barrier will have a significant cost in overhead of ordinary mutator operations.

13.7 INTERFACE TO THE COMPILER

The compiler for a garbage-collected language interacts with the garbage collector by generating code that allocates records, by describing locations of roots for each garbage-collection cycle, and by describing the layout of data records on the heap. For some versions of incremental collection, the compiler must also generate instructions to implement a read barrier or write barrier.

FAST ALLOCATION

Some programming languages, and some programs, allocate heap data (and generate garbage) very rapidly. This is especially true of programs in functional languages, where updating old data is discouraged.

The most allocation (and garbage) one could imagine a reasonable program generating is one word of allocation per store instruction; this is because each word of a heap-allocated record is usually initialized. Empirical measurements show that about one in every seven instructions executed is a store, almost regardless of programming language or program. Thus, we have (at most) $\frac{1}{7}$ word of allocation per instruction executed.

Supposing that the cost of garbage collection can be made small by proper tuning of a generational collector, there may still be a considerable cost to create the heap records. To minimize this cost, *copying collection* should be used so that the allocation space is a contiguous free region; the next free location is `next` and the end of the region is `limit`. To allocate one record of size N, the steps are:

1. Call the allocate function.
2. Test $\texttt{next} + N < \texttt{limit}$? (If the test fails, call the garbage collector.)
3. Move `next` into `result`
4. Clear $M[\texttt{next}], M[\texttt{next}+1], \ldots, M[\texttt{next}+\texttt{N}-1]$
5. $\texttt{next} \leftarrow \texttt{next} + N$
6. Return from the allocate function.

A. Move `result` into some computationally useful place.
B. Store useful values into the record.

Steps 1 and 6 should be eliminated by *inline expanding* the allocate function at each place where a record is allocated. Step 3 can often be eliminated by combining it with step A, and step 4 can be eliminated in favor of step B (steps A and B are not numbered because they are part of the useful computation; they are not allocation overhead).

Steps 2 and 5 cannot be eliminated, but if there is more than one allocation in the same basic block (or in the same *trace*, see Section 8.2) the comparison and increment can be shared among multiple allocations. By keeping `next` and `limit` in registers, steps 2 and 5 can be done in a total of three instructions.

By this combination of techniques, the cost of allocating a record – and then eventually garbage collecting it – can be brought down to about four instructions. This means that programming techniques such as the *persistent binary search tree* (page 108) can be efficient enough for everyday use.

DESCRIBING DATA LAYOUTS

The collector must be able to operate on records of all types: `list`, `tree`, or whatever the program has declared. It must be able to determine the number of fields in each record, and whether each field is a pointer.

For statically typed languages such as Tiger or Pascal, or for object-oriented languages such as Java or Modula-3, the simplest way to identify heap objects is to have the first word of every object point to a special type- or class-descriptor record. This record tells the total size of the object and the location of each pointer field.

For statically typed languages this is an overhead of one word per record to serve the garbage collector. But object-oriented languages need this descriptor-pointer in every object just to implement dynamic method lookup, so that there is no additional per-object overhead attributable to garbage collection.

The type- or class-descriptor must be generated by the compiler from the static type information calculated by the semantic analysis phase of the compiler. The descriptor-pointer will be the argument to the runtime system's `alloc` function.

In addition to describing every heap record, the compiler must identify to the collector every pointer-containing temporary and local variable, whether it is in a register or in an activation record. Because the set of live temporaries can change at every instruction, the *pointer map* is different at every point in the program. Therefore, it is simpler to describe the pointer map only at points where a new garbage collection can begin. These are at calls to the `alloc` function; and also, since any function-call might be calling a function which in turn calls `alloc`, the pointer map must be described at each function call.

The pointer map is best keyed by return addresses: a function call at location a is best described by its return address immediately after a, because the return address is what the collector will see in the very next activation record. The data structure maps return addresses to live-pointer sets; for each pointer that is live immediately after the call, the pointer map tells its register or frame location.

To find all the roots, the collector starts at the top of the stack and scans downward, frame by frame. Each return address keys the pointer-map entry that describes the next frame. In each frame, the collector marks (or forwards, if copying collection) from the pointers in that frame.

Callee-save registers need special handling. Suppose function f calls g, which calls h. Function h knows that it saved some of the callee-save registers in its frame and mentions this fact in its pointer map; but *h does not know which of these registers are pointers*. Therefore the pointer map for g must describe which of its callee-save registers contain pointers at the call to h and which are "inherited" from f.

DERIVED POINTERS

Sometimes a compiled program has a pointer that points into the middle of a heap record, or that points before or after the record. For example, the expression `a[i-2000]` can be calculated internally as `M[a-2000+i]`:

$$t_1 \leftarrow a - 2000$$
$$t_2 \leftarrow t_1 + i$$
$$t_3 \leftarrow M[t_2]$$

If the expression `a[i-2000]` occurs inside a loop, the compiler might choose to hoist $t_1 \leftarrow a - 2000$ outside the loop to avoid recalculating it in each iteration. If the loop also contains an `alloc`, and a garbage collection occurs while t_1 is live, will the collector be confused by a pointer t_1 that does not point to the beginning of an object, or (worse yet) points to an unrelated object?

We say that the t_1 is *derived* from the *base* pointer a. The pointer map must identify each *derived pointer* and tell the base pointer from which it is derived. Then, when the collector relocates a to address a', it must adjust t_1 to point to address $t_1 + a' - a$.

Of course, this means that a must remain live as long as t_1 is live. Consider the loop at left, implemented as shown at right:

```
let                                     r1 ← 100
  var a := intarray[100] of 0           r2 ← 0
                                        call alloc
                                        a ← r1
 in                                     t1 ← a − 2000
  for i := 1930 to 1990                 i ← 1930
    do f(a[i-2000])                     L1 : r1 ← M[t1 + i]
                                        call f
end                                     L2 : if i ≤ 1990 goto L1
```

If there are no other uses of a, then the temporary a appears dead after the assignment to t_1. But then the pointer map associated with the return address L_2 would not be able to "explain" t_1 adequately. Therefore, for purposes of the compiler's liveness analysis, *a derived pointer implicitly keeps its base pointer live.*

PROGRAM DESCRIPTORS

Implement record descriptors and pointer maps for the Tiger compiler.

For each record-type declaration, make a string literal to serve as the record descriptor. The length of the string should be equal to the number of fields in the record. The ith byte of the string should be p if the ith field of the record is a pointer (string, record, or array); or n if the ith field is a nonpointer.

The allocRecord function should now take the record descriptor string (pointer) instead of a length; the allocator can obtain the length from the string literal. Then allocRecord should store this descriptor pointer at field zero of the record. Modify the runtime system appropriately.

The user-visible fields of the record will now be at offsets $1, 2, 3, \ldots$ instead of $0, 1, 2, \ldots$; adjust the compiler appropriately.

Design a descriptor format for arrays, and implement it in the compiler and runtime system.

Implement a temp-map with a boolean for each temporary: is it a pointer or not? Also make a similar map for the offsets in each stack frame, for frame-resident pointer variables. You will not need to handle derived pointers, as your Tiger compiler probably does not keep derived pointers live across function calls.

For each procedure call, put a new return-address label L_{ret} immediately after the call instruction. For each one, make a data fragment of the form

```
L_ptrmap327 :   .word   L_ptrmap326   link to previous ptr-map entry
                .word   L_ret327      key for this entry
                .word   ...           pointer map for
                  ⋮                       this return address
```

and then the runtime system can traverse this linked list of pointer-map entries, and perhaps build it into a data structure of its own choosing for fast lookup of return addresses. The data-layout pseudo-instructions (`.word`, etc.) are, of course, machine dependent.

PROGRAM GARBAGE COLLECTION

Implement a mark-sweep or copying garbage collector in the C language, and link it into the runtime system. Invoke the collector from `allocRecord` or `initArray` when the free space is exhausted.

FURTHER READING

Reference counting [Collins 1960] and mark-sweep collection [McCarthy 1960] are almost as old as languages with pointers. The pointer-reversal idea is attributed by Knuth [1967] to Peter Deutsch and to Herbert Schorr and W. M. Waite.

Fenichel and Yochelson [1969] designed the first two-space copying collector, using depth-first search; Cheney [1970] designed the algorithm that uses the unscanned nodes in to-space as the queue of a breadth-first search, and also the semi-depth-first copying that improves the locality of a linked list.

Steele [1975] designed the first concurrent mark-and-sweep algorithm. Dijkstra et al. [1978] formalized the notion of tricolor marking, and designed a concurrent algorithm that they could prove correct, trying to keep the synchronization requirements as weak as possible. Baker [1978] invented the incremental copying algorithm in which the mutator sees only to-space pointers.

Generational garbage collection, taking advantage of the fact that newer objects die quickly and that there are few old-to-new pointers, was invented by Lieberman and Hewitt [1983]; Ungar [1986] developed a simpler and more efficient *remembered set* mechanism.

The Symbolics Lisp Machine [Moon 1984] had special hardware to assist with incremental and generational garbage collection. The microcoded memory-fetch instructions enforced the invariant of Baker's algorithm; the microcoded memory-store instructions maintained the remembered set for generational collection. This collector was the first to explicitly improve locality of reference by keeping related objects on the same virtual-memory page.

As modern computers rarely use microcode, and a modern general-purpose processor embedded in a general-purpose memory hierarchy tends to be an order of magnitude faster and cheaper than a computer with special-purpose instructions and memory tags, attention turned in the late 1980s to algorithms that could be implemented with standard RISC instructions and standard virtual-memory hardware. Appel et al. [1988] use virtual memory to implement a read barrier in a truly concurrent variant of Baker's algorithm. Shaw [1988] uses virtual memory *dirty bits* to implement a write barrier for generational collection, and Boehm et al. [1991] make the same simple write barrier serve for concurrent generational mark-and-sweep. Write barriers are cheaper to implement than read barriers, because stores to old pages are rarer than fetches from to-space, and a write barrier merely needs to set a dirty bit and continue with minimal interruption of the mutator. Sobalvarro [1988] invented the card marking technique, which uses ordinary RISC instructions without requiring interaction with the virtual-memory system.

Appel and Shao [1996] describe techniques for fast allocation of heap records and discuss several other efficiency issues related to garbage-collected systems.

Branquart and Lewi [1971] describe pointer maps communicated from a compiler to its garbage collector; Diwan et al. [1992] tie pointer maps to return addresses, show how to handle derived pointers, and compress the maps to save space.

Appel [1992, Chapter 12] shows that compilers for functional languages must be careful about closure representations; using simple static links (for example) can keep enormous amounts of data reachable, preventing the collector from reclaiming it.

Boehm and Weiser [1988] describe *conservative collection*, where the compiler does not inform the collector which variables and record-fields contain pointers, so the collector must "guess." Any bit-pattern pointing into the allocated heap is assumed to be a possible pointer and keeps the pointed-to record live. However, since the bit-pattern might really be meant as an integer, the object cannot be moved (which would change the possible integer), and some garbage objects may not be reclaimed. Wentworth [1990] points out that such an integer may (coincidentally) point to the root of a huge garbage data structure, which therefore will not be reclaimed; so conservative collection will occasionally suffer from a disastrous space leak. Boehm [1993] describes several techniques for making these disasters unlikely: for example, if the collector ever finds an integer pointing to address X that is not a currently allocated object, it should *blacklist* that address so that the allocator will never allocate an object there. Boehm [1996] points out that even a conservative collector needs some amount of compiler assistance: if a derived pointer can point outside the bounds of an object, then its base pointer must be kept live as long as the derived pointer exists.

Page 509 discusses some of the literature on improving the cache performance of garbage-collected systems.

Cohen [1981] comprehensively surveys the first two decades of garbage-collection research; Wilson [1997] describes and discusses more recent work. Jones and Lins [1996] offer a comprehensive textbook on garbage collection.

EXERCISES

***13.1** Analyze the cost of mark-sweep versus copying collection. Assume that every record is exactly two words long, and every field is a pointer. Some pointers may point outside the collectible heap, and these are to be left unchanged.

a. Analyze Algorithm 13.6 to estimate c_1, the cost (in instructions per reachable word) of depth-first marking.

b. Analyze Algorithm 13.3 to estimate c_2, the cost (in instructions per word in the heap) of sweeping.

c. Analyze Algorithm 13.9 to estimate c_3, the cost per reachable word of copying collection.

d. There is some ratio γ so that with $H = \gamma R$ the cost of copying collection equals the cost of mark-sweep collection. Find γ.

e. For $H > \gamma R$, which is cheaper, mark-sweep or copying collection?

13.2 Run Algorithm 13.6 (pointer reversal) on the heap of Figure 13.1. Show the state of the heap, the `done` flags, and variables `t`, `x`, and `y` at the time the node containing 59 is first marked.

***13.3** Assume `main` calls `f` with callee-save registers all containing 0. Then `f` saves the callee-save registers it is going to use; puts pointers into some callee-save registers, integers into others, and leaves the rest untouched; then calls `g`. Now `g` saves some of the callee-save registers, puts some pointers and integers into them, and calls `alloc`, which starts a garbage collection.

a. Write functions `f` and `g` matching this description.

b. Illustrate the pointer maps of functions `f` and `g`.

c. Show the steps that the collector takes to recover the exact locations of all the pointers.

****13.4** Every object in the Java language supports a `hashCode()` method that returns a "hash code" for that object. Hash codes need not be unique – different objects can return the same hash code – but each object must return the same hash code every time it is called, and two objects selected at random should have only a small chance of having the same hash code.

The Java language specification says that "This is typically implemented by converting the address of the object to an integer, but this implementation technique is not required by the Java language."

Explain the problem in implementing `hashCode()` this way in a Java system with copying garbage collection, and propose a solution.