Chapter 4

Automatic Memory Management

Contents

To implement μScheme, we must continually allocate fresh locations. How can we provide an arbitrary supply of locations? Virtual memory is limited, and if we call `malloc` over and over, eventually it will fail. If we rely on the programmer to free a location when it is no longer needed, as in C and C++, we risk memory leaks, locations that are freed multiple times, and the use of locations after they are freed (so-called dangling-pointer errors). The last two mistakes can make a program crash, or worse, silently produce wrong answers. In μScheme, full Scheme, Java, and other safe languages, such mistakes are impossible. μScheme is safe because the *implementation* of μScheme, not the μScheme programmer, figures out when it is safe to reuse a location. The Scheme report puts it this way:

> No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation (Kelsey, Clinger, and Rees 1998, page 3).

In this chapter, we demonstrate the techniques used to find and reuse locations that are no longer needed by μScheme programs.

Memory-management techniques might seem like matters for implementors, not for language designers, but memory management enables a critically important feature: the ability to create closures, cons cells, and other objects without having to say when the memory they occupy must be reclaimed. Without automatic memory management, programming with higher-order functions would be nearly impossible, and object-oriented systems like Smalltalk (Chapter 10) would be much more complex. Safety guarantees, like those provided with Java, would be impossible. Automatic memory management is an essential topic because it underlies all civilized programming languages, and it makes many more choices available to the language designer.

This chapter explains the basic ideas of reachability and roots that make it possible to manage memory automatically. It focuses on *garbage collection*, the most widely used method of memory management, and its implementation in μScheme. Section 4.7 briefly discusses reference counting, an alternative method.

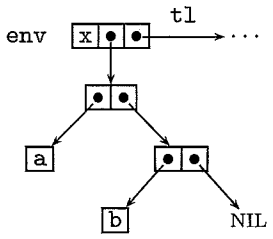
4.1 What garbage is and where it comes from

The μScheme interpreter uses `Value` objects in two ways. Temporary values, such as the head value `v` produced during the execution of `evallist`, are stored in local variables, i.e., locations on the C stack. This stack allocation is possible because the *lifetime* (sometimes called *extent*) of a temporary value is known. In `evallist`, for example, we keep the value in `v` until a copy of it is passed to `mkVL`, after which the copy on the stack is not used again.

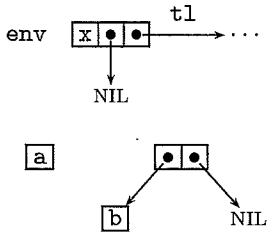
But some `Value` objects are allocated on the C heap, with `malloc`, exactly because we can't predict how long they will live. For example, if a μScheme function returns a cons cell or a closure, or if it stores one of these objects in a global variable, it is in general undecidable whether that object will be needed in a future computation. In C, of course, it would be up to the C programmer to call `free` at an appropriate moment, or never to call `free` and to risk running out of memory. In Scheme, it is up to the system to decide when to return unneeded memory to the heap for re-use. This decision not only relieves the Scheme programmer of the burden of deciding when to call `free`; it also gives the Scheme programmer the freedom to use primitive constructs like `cons` and `lambda` *without having to be aware of whether they allocate memory.* The ability to create language primitives that might allocate gives tremendous power to a language designer.

How can the system know if an object is unneeded and can be reused? At any given time, some `Value` objects may have become *unreachable*, which is to say there is no way to get to them. Here are some examples of how objects become unreachable:

- Consider the evaluation of `(val x '(a b))` at the top level. Five `Value` objects are allocated, as well as an `Env` link for the top-level environment. In this picture, the `Env` object is labelled `env`; the unlabelled objects are `Values`.



If the very next expression evaluated is `(set x '())`, we get:



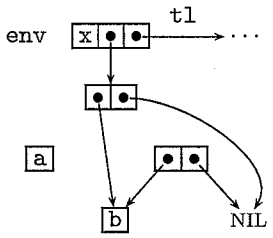
We have overwritten the location associated with `x`, so now the four locations holding the atom `a`, the cons cell `(b)`, the atom `b`, and NIL are cut off—unreachable—and can never again be relevant to the computation. They can be reused to hold future values.

- Suppose instead of setting `x` to NIL, we set it to `(cdr x)`.

175 ⟨*transcript* 175⟩≡

```
-> (val x '(a b))
-> (set x (cdr x))
```

We copy the second cons cell over the location containing the first. The picture now is:



The locations containing the atom `a` and the original copy of the second cell of `(a b)` are now unreachable.

Locations holding unreachable objects are also called *garbage*. In a *garbage-collected* system, the program allocates objects from the heap until it runs out. When the heap is exhausted, the computation is halted while a garbage collector makes unreachable objects available to satisfy future allocation requests. In a simple garbage collector, the overhead is very bursty; most allocations are fast, but an allocation that triggers the garbage collector makes the whole program pause, sometimes for a second or more. Sophisticated garbage collectors run more frequently, in shorter bursts, or even concurrently, on another processor, in order to make the pauses imperceptibly short. As we discuss below, garbage-collected systems can easily trade space for time; by enlarging the heap, one can make the collector run less often.

Garbage collection reduces a program's total memory requirement to the memory occupied by *reachable* objects; even a computation that allocates an enormous number of objects may have relatively few reachable objects at any given time. Recycling unreachable objects can dramatically reduce use of memory.

Garbage collection has another pleasant property—as we show in this chapter, it is easy to integrate a garbage-collected heap into existing code; one simply replaces `malloc` with a new allocator, and `free` can be implemented as a no-op.

There are many variations on garbage collection, but all involve scanning the *roots* of the system and using these roots to find the reachable objects. Every variation is founded on one of the two methods we discuss: *mark-and-sweep* and *copying* collection.

An alternative method not based on reachability is *reference counting*. The idea is that each object includes a count of the number of pointers to that object; when the count goes to zero, the object can be put on a free list. Reference counting cannot reclaim cyclic garbage; an unreachable circular list has no objects with zero counts, so the objects never get reclaimed. For this reason and others, garbage collection is normally preferable to reference counting; Section 4.7 discusses the details.

Some authors use the term "garbage collection" to refer to any strategy for automatic memory management, including reference counting. We reserve the term for systems with a true garbage collector.

4.2 Garbage-collection basics

The garbage collector and allocator cooperate to implement a *managed heap*. Although most computer scientists emphasize the garbage collector, it's a mistake to overlook the allocator; the two work as a team, and the only way to understand the performance of a managed heap is to consider both.

4.2.1 Performance

There are many measures on which different memory-management strategies might be compared; the following three are the most important in practice:

Allocation cost. How long does it take to allocate an object?

Collector overhead. How much additional work, per object allocated, does it take to run the garbage collector?

Memory overhead. How much additional memory, per object, is needed to hold the data structures that support the managed heap?

The first two measures combine to give the cost per allocation, measured in time. The third measure gives the space overhead of garbage collection, e.g., the ratio of the size of the whole heap to the size of the program's data. In garbage-collected systems, space can be traded for time by adjusting the size of the heap, H; we can lower the cost per allocation by using a larger heap.

A fourth measure is also worth keeping in mind:

Pause time. How long does the program pause while the garbage collector is running?

This question is important for programs that must interact with users, or for servers that must answer requests in a timely fashion. There is another tradeoff available here; if one is willing to increase the cost per allocation, one can decrease pause times. State-of-the-art real-time collectors have pause times below 60 msec. Since any response time under 100 msec appears instantaneous to a human being, these pause times are very effective in interactive applications.

4.2.2 Reachability and roots

As shown above, heap-allocated objects can become irrelevant as a program runs. Moreover, they often do so in an unpredictable manner and at unpredictable times. A heap object may be used immediately and never needed again, or it may be bound into a global environment and live for the rest of the session. In fact, even though computation is deterministic, it is undecidable in general whether any particular heap object can possibly matter to any future computation. Luckily, an excellent approximation is available.

We assume that if the interpreter can start at any active environment, and it can follow pointers to reach an object, then that object might be used in a future computation, and we call the object *live data*. Any object that cannot be so reached is *unreachable* and can safely be reclaimed.

We define the live data by induction. The base case is given by a set of *roots*, which are enumerated below; roots are live by definition. The induction step says that any object pointed to by a live object is itself a live object. The garbage collector can therefore find *all* live data by starting at the roots and tracing pointers until it has found all the objects. We call the amount of live data L.

In compiled code, the set of roots is fairly straightforward. Roots are found in three places:

- Global variables from which heap-allocated objects might be reachable
- Local variables and formal parameters, anywhere on the call stack, from which heap-allocated objects might be reachable
- Hardware registers from which heap-allocated objects might be reachable

Things are similar in our μScheme interpreter. We needn't worry about machine registers, but we do need to worry about global variables, formal parameters, and local variables. Since the number of live local variables changes as the interpreter calls C procedures and those procedures return, the set of roots changes as well. We therefore need a way to identify the roots *dynamically*.

Tricolor marking

The collection of allocated objects, together with the pointers between them, form a graph. *Tricolor marking* of this graph is a conceptual tool that can help you understand how pointer tracing finds live objects. We imagine that each object is colored as follows:

- A *white* object has not been visited by the garbage collector.
- A *gray* object has been visited, but not all of its successors have been visited.
- A *black* object has been visited, and so have all of its successors.

The *coloring invariant* is this: no black object ever points to a white object; black objects point only to gray objects or to other black objects. Both mark-and-sweep and copying collectors traverse the graph of objects this way:

```
color all the roots gray
while there is a gray object remaining
    choose a gray object
    if the object has a white successor
        color that successor gray
    else
        color the gray object black
```

This algorithm maintains the coloring invariant. There are finitely many objects, every gray object eventually becomes black, and no black object changes color. Thus the algorithm terminates. When it does so, the heap contains only black and white objects. The black objects are exactly those reachable from the roots: the live data. The white objects are garbage and can be reused.

Of course, this algorithm is highly underspecified; it does not say which gray object to pick next or which white successor to color gray. Different collectors make these choices differently; for example, a typical mark-and-sweep collector visits objects depth-first, while a typical copying collector visits breadth-first. Mark-and-sweep and copying collectors also have different conditions that correspond to the three colors.

4.2.3 Heap growth

In addition to recovering and reusing unreachable objects, the garbage collector must also manage the size of the heap. At minimum, the collector must ensure that the heap is large enough to hold all the program's live data. For example, if a program computes a list of all permutations of a list of length n, just holding the answer requires $n!$ lists of n elements, so at minimum $L \geq nn!$. If there are not that many objects available on the heap, the program is doomed to fail.

Older collectors didn't worry about the size of the heap: in the very first garbage-collected systems, a computer ran one program at a time, and that program ran in all of physical memory. The only really difficult part of garbage collection was to get reasonable performance in that awkward regime where the heap is just barely larger than the amount of live data. For example, if $H - L = 1$, then the garbage collector will have to be run after *every* allocation, which is very expensive.

In today's systems, garbage collectors run on virtual-memory machines, and it suffices for the garbage collector to keep the heap small enough so that virtual memory performs well, while keeping it large enough so that the overhead of garbage collection is not too high. A typical strategy is for the collector to start out with a fairly small heap, then enlarge it in response to the growth of live data. To amortize the costs of enlarging the heap, the collector may enlarge it in fairly big chunks, e.g., many virtual-memory pages.

How much should the heap grow? A good, simple rule of thumb is to maintain a roughly constant *ratio of heap size to live data*, which we write $\gamma = \frac{H}{L}$. If you prefer, you can think of $\frac{1}{\gamma}$, which is the fraction of heap memory that is used to hold the program's data. As we see below, γ bounds the fraction of memory that holds no data, and it also is a good predictor of the overhead of garbage collection. Clients can adjust space-time tradeoffs by adjusting γ. If γ is too small, there are too few allocations between collections, and the overhead of collection can get very high. Modern mark-and-sweep collectors perform very well when $\gamma > 2$; copying collectors do better with $\gamma > 3$. It is difficult to make apples-to-apples comparisons, but standard implementations of `malloc` and `free` appear to require γ in the range of 1.1 to 1.4 (Wilson et al. 1995; Johnstone and Wilson 1998).

It is not unusual to reduce garbage-collection overhead by pushing γ up to 5 or 7. A larger γ requires a larger heap, which means that more memory holds no data, but it is unfair to think of the memory as wasted. The memory is used to reduce time: time spent collecting, time spent worrying whether objects in a program need to be freed explicitly, or time spent tracking down memory leaks, double frees, and dangling-pointer errors. Other dynamic-allocation systems also trade space for time; Grunwald and Zorn (1993) show that fast implementations of `malloc` may require heaps 1.4 to 2 times larger than a simple first-fit implementation.

4.3 The managed heap in μScheme

In a real Scheme system, all dynamically allocated objects are on the managed heap. In μScheme, for simplicity, we manage only allocated values, those obtained through `allocate`. Values of types `Exp`, `Explist`, and `Namelist` get allocated so slowly—at the speed of human typing—that it is not worth the trouble of avoiding their dynamic allocation, for expository purposes. Values of other types, such as `Valuelist` and `Env`, are allocated at greater rates, but managing them would clutter the exposition. Real garbage collectors deal with all allocations using the same techniques we use for `Values`.

4.3.1 Stack-allocated values vs. heap-allocated values

We distinguish stack-allocated values, which the C compiler allocates and reclaims automatically, from heap-allocated values, which are allocated by explicit calls to `allocate` and whose reclamation is the job of the garbage collector.

The simple rule is this: if a pointer to a `Value` exists in another data structure, such as another `Value`, an `Exp`, or an `Env`, the pointer must point to a location allocated on the heap. Structures may not contain pointers to `Values` allocated on the stack.

4.3.2 Reachability and roots

Even though we manage only the allocation and reuse of values, we have to consider roots of other types, because for example, an environment might point to an allocated value, making it reachable. As a first step toward identifying roots, we list the *types* of C values from which heap-allocated locations containing value objects might be reachable. A value of any of the following types might contain a pointer from which a heap-allocated `Value` is reachable. We divide them into two categories.

A. Any structure or pointer that might contain a pointer to a `Value` allocated on the heap is a potential root of category A. These types and their associated pointers are:

   - `Value`, through `u.pair.car` and `u.pair.cdr`
   - `Value*`, which points directly to a heap-allocated object
   - `Env`, through `loc`
   - `Exp`, through `u.val`

B. Any structure or pointer that might contain or point to a potential root of category A is itself a potential root of category B. To avoid infinite regress, any structure or pointer that might contain or point to a potential root of category B is also a potential root of category B.

   - `Valuelist`s, because the `hd` field contains a `Value`
   - `Lambda`, through `body`
   - `Value` again, through `u.closure`
   - `Exp` again, through `u.lambda` or the many `Exp*`s it contains
   - `Env`, through `tl`
   - `Explist`s, because the `hd` field contains an `Exp`
   - `Valuelist` again, because the `tl` field points to a `Valuelist`
   - `Def`s, because they point to `Exp`s

The information above shows not only from which types of value a heap object might be reachable but also which pointers to follow from such a value to reach the heap object. Accordingly, for each type above, we have written a procedure that follows pointers and marks heap-allocated values. These procedures appear in Sections 4.4.2 and C.1.

Any value of any of the types named above is potentially a root if it could be live during an allocation. The set of roots therefore includes:

1. The global environment.
2. The current definition being evaluated.
3. Local variables and actual parameters of any function that calls a function that could allocate.

4.3.3 Interface to the managed heap

This interface specifies how the heap is controlled and what invariants clients must establish and maintain in order to interact with the heap. The interface comprises two parts: allocation and root-tracking.

Because potential roots include local variables of C functions such as `eval`, the set of roots changes as the interpreter executes. In a compiled system, the garbage collector is in cahoots with the compiler, which knows where all the local variables are. In fact, the compiler knows everything about the layout of the activation records and the call stack. A compiler can therefore arrange for a garbage collector to be able to find roots easily, with no added overhead during "real" computation. Since our garbage collector is *not* in cahoots with the C compiler, we need some other way of finding roots. What we do is maintain an explicit *root stack*, in addition to the C call stack, and we put extra code in our C procedures to push roots onto and pop roots off of the root stack.

181a ⟨*function prototypes for μScheme* 181a⟩≡ (125c) 181b▷
```
void pushroot(Root r);
void poproot (Root r);
```

It is a checked run-time error if the value passed to `poproot` doesn't match the corresponding `pushroot`.

We make the allocator independent of initialization; chunk 182d shows how to use `allocloc` to implement `allocate`.

181b ⟨*function prototypes for μScheme* 181a⟩+≡ (125c) ◁181a 182a▷
```
Value *allocloc(void);
```

The allocator and root stack are related by this precondition: Clients may call `allocloc` only when *all objects that could lead to live values have been pushed on the root stack.* The copying collector's implementation of `allocloc` also requires that, when called, *all pointers to allocated values must be reachable from the root stack*, so that they can be updated when the values move. Section C.4 explains the programming techniques we use to ensure that these preconditions are satisfied.

What is left is the representation of roots. We need a *pointer* to any root of category A, in order to update internal pointers to heap-allocated objects when those objects move. In addition to `Env` and `Exp`, these roots include `Value` objects allocated on the C stack (tag `STACKVALUEROOT`) as well as pointers to `Value` objects allocated on the heap (tag `HEAPLOCROOT`). Although it might seem that both kinds of roots are "just values," the distinction is necessary for both mark-and-sweep and copying collectors.

- In a mark-and-sweep collector, heap-allocated objects have mark bits, but stack-allocated objects do not.
- In a copying collector, heap-allocated objects move, but stack-allocated objects do not.

Here are all the kinds of roots, of both categories.

181c ⟨*root.t* 181c⟩≡
```
Root = STACKVALUEROOT (Value*)      // pointer to stack-allocated object
     | HEAPLOCROOT    (Value**)     // pointer to pointer to heap-allocated object
     | ENVROOT        (Env*)
     | EXPROOT        (Exp*)
     | EXPLISTROOT    (Explist*)
     | VALUELISTROOT  (Valuelist*)
     | DEFROOT        (Def*)
```

The double indirection for `Value**` is needed, because if an object on the heap moves, we need to update a `Value*`. The indirections for the last five kinds of roots are not strictly necessary; they are there for convenience. For example, if `&env` is on the root stack, we can safely execute `env = bindalloc(..., env)` without having to touch the root stack. If `env`, not `&env`, were on the stack, we would have to pop the old copy and push the new one after the assignment.

There are a few more operations that affect the root stack. Clients may use `resetrootstack` to clear the root stack, e.g., in case an error unwinds the C stack.

182a ⟨*function prototypes for μScheme* 181a⟩+≡ (125c) ◁181b 182b▷
```
void resetrootstack(void);
```

Collectors need to get at the roots on the stack. Rather than define a procedural interface, we expose the representation. The invariant is that for all $0 \leq$ `i` $<$ `rootstacksize`, `rootstack[i]` is a live root. The representation is *read only*; it is an *unchecked* run-time error for a client to modify `rootstack` or `rootstacksize`.

182b ⟨*function prototypes for μScheme* 181a⟩+≡ (125c) ◁182a 182c▷
```
extern Root *rootstack;
extern int rootstacksize;
```

Finally, the interface provides a routine to dump the root information; `printroots` may help you debug code for the Exercises.

182c ⟨*function prototypes for μScheme* 181a⟩+≡ (125c) ◁182b
```
void printroots(void);
```

Section C.4 shows how we have modified the evaluator to keep track of roots, and Section C.5 shows the implementation of the root stack. The code is not representative of a real system, but if you are curious about the root stack, Sections C.4 and C.5 are the places to look.

4.3.4 Implementation of the μScheme allocation interface

As a small illustration of the use of the root stack, we show how to use the heap interface to implement the allocation interface defined in Chapter 3. The copying and mark-and-sweep collectors use different implementations of `allocloc`, but they share this implementation of `allocate`.

182d ⟨*loc.c* 182d⟩≡ 183a▷
```
#include "all.h"

Value* allocate(Value v) {
    Value *loc;

    pushroot(mkStackvalueroot(&v));
    loc = allocloc();
    assert(loc != NULL);
    poproot(mkStackvalueroot(&v));
    *loc = v;
    return loc;
}
```

allocloc 181b
mkStackvalueroot 4
poproot 181a
pushroot 181a

The interesting part is that `allocloc` might make objects on the heap move, so before we call `allocloc`, we have to put `v` on the root stack. This way, if `v` is a pair, its `car` and `cdr` fields are sure to be updated correctly.

One should call `initallocate` only at startup time. It pushes an environment onto the root stack, and it sets the environment to `NULL` so there won't be a pointer to an uninitialized environment on the root stack.

183a ⟨*loc.c* 182d⟩+≡ ◁182d 183b▷

```
void initallocate(Env *envp) {
    pushroot(mkEnvroot(envp));
    *envp = NULL;
}
```

Function `resetallocate` simply adjusts the stack.

183b ⟨*loc.c* 182d⟩+≡ ◁183a

```
void resetallocate(Env *envp) {
    resetrootstack();
    pushroot(mkEnvroot(envp));
}
```

4.4 Mark-and-sweep garbage collection

The oldest method of garbage collection is called *mark-and-sweep* or *mark-scan*. Every object is associated with a bit called the *mark bit*. The allocator gets available locations from a linked list called the *free list*. When all the locations on the free list have been allocated, the allocator calls the garbage collector. It is possible to write a very simple allocator:

1. As long as the free list is empty, call the collector.
2. Remove an object from the free list and return it.

When called, the collector proceeds in three phases:

1. Unmark (i.e., clear the mark bit associated with) *all objects* in the heap, whether reachable or not.
2. Mark (i.e., set the mark bit associated with) all reachable objects. This phase involves a depth-first traversal of the heap, starting from the roots.
3. Sweep all objects in the heap, placing unmarked objects on the free list.

The problem with this naïve approach is that the unmark and sweep phases require visiting the entire heap, which is slow. The program pauses for a long time while the collector runs. The problem is worsened because the heap is unlikely to fit in the cache.

A better algorithm is to let the *allocator* do the unmarking and sweeping. Unmarking and sweeping still require visiting the entire heap, but the visit is spread out over a large number of allocation requests. Since the collector only has to mark, its running time drops significantly. Using this algorithm, the allocator keeps a pointer into the managed heap, advancing the pointer one or more objects until it encounters one that can be reused. We sketch the beginnings of such an allocator below; Exercise 3 elaborates.

atexit *B*
mkEnvroot *A*
pushroot 181a
resetrootstack 182a

4.4.1 Prototype mark-and-sweep allocator for μScheme

We present code for a prototype allocator; the Exercises explain how to extend this allocator into a full mark-and-sweep system.

184a ⟨*ms.c* 184a⟩≡ 185a▷
```
#include "all.h"

⟨private declarations for mark-and-sweep collection 184b⟩
```

A mark-and-sweep collector needs to associate a mark bit with each heap location. For simplicity, we make no attempt to pack mark bits densely; we simply wrap each `Value` in another structure, which holds a single mark bit, `live`. By placing the `Value` at the beginning, we ensure that it is safe to cast between values of type `Value*` and type `Mvalue*`.

184b ⟨*private declarations for mark-and-sweep collection* 184b⟩≡ (184a) 184c▷
```
typedef struct Mvalue Mvalue;
struct Mvalue {
    Value v;
    unsigned live:1;
};
```

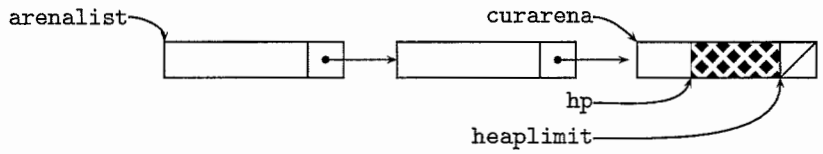
We put these wrapped values into *arenas*. A single arena holds a contiguous array of objects; the entire heap is formed by chaining arenas into linked lists. The arena is the unit of heap growth; when the heap is too small, we call `malloc` to add one or more arenas to the heap.

184c ⟨*private declarations for mark-and-sweep collection* 184b⟩+≡ (184a) ◁184b 184d▷
```
enum growth_unit { GROWTH_UNIT = 24 };
typedef struct Arena *Arena;
struct Arena {
    Mvalue pool[GROWTH_UNIT];
    Arena tl;
};
```

We use the `tl` field to chain arenas on a linked list; the head of that list is called `arenalist`. The "heap pointer" `hp` points to the next `Mvalue` to be allocated. The pointer `heaplimit` points to the first `Mvalue` *after* the current arena, `curarena`.

184d ⟨*private declarations for mark-and-sweep collection* 184b⟩+≡ (184a) ◁184c 186a▷
```
Arena arenalist, curarena;
Mvalue *hp, *heaplimit;
```

Here's a picture of what the arenas look like:



White areas have been allocated, and areas marked in black diamonds are available for allocation. Arenas except the current one are entirely used. The number of unallocated cells in the current arena is `heaplimit - hp`.

The function `makecurrent` makes `curarena`, `hp`, and `heaplimit` point to an arena from which no locations have yet been allocated.

185a ⟨*ms.c* 184a⟩+≡ ◁184a 185b▷

```
static void makecurrent(Arena a) {
    assert(a != NULL);
    curarena = a;
    hp = &a->pool[0];
    heaplimit = &a->pool[GROWTH_UNIT];
}
```

When the heap grows, it grows by one arena at a time. We allocate new arenas with `calloc` so that the mark bits are zeroed. It is a checked run-time error to call `addarena` except when `arenalist` is `NULL` or when `curarena` points to the last arena in the list.

185b ⟨*ms.c* 184a⟩+≡ ◁185a 186b▷

```
static void addarena(void) {
    Arena a = calloc(1, sizeof(*a));
    assert(a != NULL);

    if (arenalist == NULL) {
        arenalist = a;
    } else {
        assert(curarena != NULL && curarena->tl == NULL);
        curarena->tl = a;
    }
    makecurrent(a);
}
```

We provide a prototype allocator that never collects garbage; when it runs out of space, it just adds a new arena. Exercise 3 discusses how to replace this allocator with one that supports garbage collection.

185c ⟨*ms.c* [[**prototype**]] 185c⟩≡ 186d▷

```
Value* allocloc(void) {
    if (hp == heaplimit)
        addarena();
    assert(hp < heaplimit);
    return &(hp++)->v;
}
```

4.4.2 Marking heap objects in μScheme

If we visualize the heap as a directed graph containing objects of different types, the collector's marking phase is essentially a depth-first search. Starting at the roots, for each object, we visit the objects it points to. When we visit a `Value` allocated on the heap, we set the mark bit. If we visit such a `Value` and the mark is already set, we return immediately; this test guarantees that the mark phase terminates even if there is a cycle on the heap.

`arenalist` 184d
`calloc` *B*
`curarena` 184d
`heaplimit` 184d
`hp` 184d

The search itself is straightforward; there is one procedure for each type of object to be visited. Function `visitroot` looks at the tag of a root and calls the appropriate visiting procedure.

186a ⟨*private declarations for mark-and-sweep collection* 184b⟩+≡ (184a) ◁184d

```
static void visitheaploc      (Value *loc);
static void visitstackvalue   (Value v);
static void visitvaluechildren(Value v);
static void visitenv          (Env env);
static void visitvaluelist    (Valuelist vl);
static void visitexp          (Exp exp);
static void visitexplist      (Explist el);
static void visitdef          (Def def);
static void visitroot         (Root r);
```

Writing these procedures is mostly straightforward. Here, for example, is the code to visit an `Env` node.

186b ⟨*ms.c* 184a⟩+≡ ◁185b 186e▷

```
static void visitenv(Env env) {
    for (; env; env = env->tl)
        visitheaploc(env->loc);
}
```

In order for this function to work, we have to expose the representation of environments. In Chapter 3, this representation is private.

186c ⟨*structure definitions for μScheme* 186c⟩≡ (125c)

```
struct Env {
    Name name;
    Value *loc;
    Env tl;
};
```

The most subtle aspect of the visiting functions is that they must distinguish values allocated on the heap from those allocated on the stack. A heap-allocated value is embedded inside an `Mvalue` and has a mark bit, but a stack-allocated value is not in an `Mvalue` and has no mark bit. For both kinds of values, we need to visit the children, but only for a heap-allocated value do we set a mark bit.

186d ⟨*ms.c* [[**prototype**]] 185c⟩+≡ ◁185c

```
static void visitheaploc(Value *loc) {
    Mvalue *m = (Mvalue*)loc;
    if (!m->live) {
        m->live = 1;
        visitvaluechildren(m->v);
    }
}
```

visitvaluechildren 187

186e ⟨*ms.c* 184a⟩+≡ ◁186b 187▷

```
static void visitstackvalue(Value v) {
    visitvaluechildren(v);
}
```

Function `visitvaluechildren` works the same whether a value is allocated on the stack or the heap.

187 ⟨*ms.c* 184a⟩+≡ ◁186e

```
static void visitvaluechildren(Value v) {
    switch (v.alt) {
    case NIL:
    case BOOL:
    case NUM:
    case SYM:
    case PRIMITIVE:
        return;
    case PAIR:
        visitheaploc(v.u.pair.car);
        visitheaploc(v.u.pair.cdr);
        return;
    case CLOSURE:
        visitexp(v.u.closure.lambda.body);
        visitenv(v.u.closure.env);
        return;
    default:
        assert(0);
        return;
    }
}
```

The implementations of the remaining marking procedures appear in Section C.1.

4.4.3 Performance

In a garbage-collected system, it is extremely difficult to predict the cost of any single allocation, in part because we never know which allocation might trigger a garbage collection or how much data might be live at that time. An *amortized* analysis, in which we consider a sequence of allocations, is more tractable. In such an analysis, we consider an entire garbage-collection cycle. How much does it cost to run the garbage collector, then make N calls to the allocator, right up to just before the next time the collector is run? The key to the analysis is to realize that in one cycle, the allocator sweeps all H cells, of which L are marked live, so $N = H - L$.

We make one important simplifying assumption: we assume that the heap is in "steady state," i.e., H, L, and γ do not change from one cycle to the next. Real heaps often grow and shrink wildly, so this assumption seldom holds in practice, but it is still pretty good at helping to predict and compare the costs of different garbage-collection techniques.

`visitexp` 622

Allocation cost. For mark-and-sweep, the cost per allocation is a constant (for finding and returning an unmarked cell), plus some fraction of the sweeping and unmarking cost for the whole heap. As shown in Exercises 6 and 9, the sweeping and unmarking cost is usually small on average, and with high probability, it is bounded by a small constant.

Collector overhead. The total work done in the garbage collector is proportional to the size of the live data, not the size of the entire heap. The work itself is depth-first search and setting mark bits, which is not too expensive, depending on exactly where the mark bits are stored. To compute work per allocation, divide by the number of allocations.

Memory overhead. There are five memory overheads for a mark-and-sweep system:

- We need one mark bit per object. If there is not already a bit available in the object header, the mark bits can be pushed off into a separate bitmap.
- We need information that tells where to find pointers in heap objects. Our μScheme interpreter uses the same `ty` field that is used to identify the type of a value, so there is no additional overhead here. Real systems use a variety of tricks to keep this overhead low.
- The mark-and-sweep system requires $\gamma > 1$ to perform well. While γ can be adjusted through a wide range, letting γ get too close to 1 would result in poor performance. The necessary "cushion" may be considered a memory overhead.
- When used to allocate objects of different sizes, mark-and-sweep systems may suffer from *fragmentation*, i.e., they may have chunks of free memory that are too small to satisfy allocation requests. The classic analyses of dynamic memory allocators apply equally well to mark-and-sweep allocators (Knuth 1973).
- The mark-and-sweep collector needs a stack to traverse the live data. Our implementation uses the C stack, which is typical of today's collectors. Some older systems did not call the garbage collector until memory was completely exhausted, without room for a stack. Deutsch and, independently, Schorr and Waite found a way to do the necessary traversals without using a stack; see references under "Further Reading."

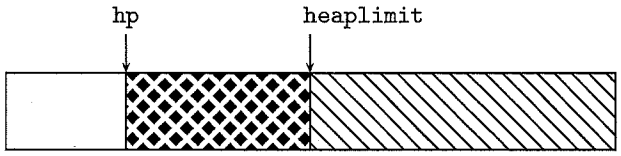
Pause time. Because the allocator must sweep past marked objects, it is possible to have a significant pause at allocation time, but on average allocation is fast, and the worst case is fast with high probability.

When a garbage collection is triggered, the whole system must pause long enough for the collector to mark live data. If there is a lot of live data, such a pause may be too long for real-time response, but it is still much better than the naïve version, which requires the collector both to unmark and to sweep the heap. Collectors that run in time proportional to the amount of live data are effective for many applications.

4.5 Copying garbage collection

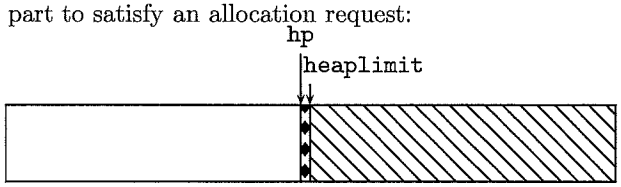
The copying, or *stop-and-copy*, method of garbage collection represents a dramatic trade-off of space for time. By using only half the heap at a time—sacrificing, in other words, half of memory—a copying system is able to keep its free space in one contiguous segment, making allocation blindingly fast. The idea is this:

- Divide the heap into two equal *semispaces*, and use one at a time. The half in use, called *from-space*, is itself divided into two *unequal* parts; the *heap pointer* marks the boundary between them. The first part contains objects that have been allocated; the second contains memory that is available for allocation. The allocator simply advances the heap pointer, so new memory is allocated sequentially from the available part until it runs out. There is no free list. This diagram shows a heap in a copying system.



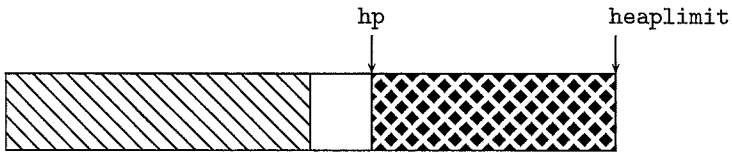
  Within from-space, on the left, the allocated area is white and the unallocated area is filled with black diamonds; the heap pointer `hp` separates the two. The limit pointer `heaplimit` marks the end of the unallocated area. The striped area is not used during allocation.

  As memory is allocated, we increment `hp`. The white part grows and the black-diamond part shrinks, until all of *from-space* contains allocated objects, and there is not enough black-diamond part to satisfy an allocation request:



- When the available memory in from-space is used up, we switch to the other semispace, called *to-space*. Before the switch, we *copy* all the reachable objects from from-space to to-space. Because not all objects are reachable, we have room left over in to-space to satisfy future allocation requests. We therefore "flip" the two spaces, and continue executing in to-space; the allocator takes new memory from the first location above the copied objects.

  After the flip, the heap looks like this:



  About 20% of objects have survived the collection, and 20% of a semispace is 10% of the heap, so $\gamma \approx 10$. When the survival rate is only 20%, a copying collector performs very well.

4.5.1 How copying collection works

A copying collector has something in common with the "marshallers" used to send structured data in distributed systems; both components move data from one place to another.[1] To preserve sharing and cycles, a copying collector must copy each live object exactly once.

One way to keep track of which objects have been copied would be to associate a bit with each one, just as we do for mark and sweep. But once an object has been copied, the space it formerly occupied is just sitting around, unused. We can therefore reuse that space immediately, by overwriting each copied object with a *forwarding pointer*. The forwarding pointer indicates that the object has already been copied, and it also points to the location of the object in to-space. In our μScheme system, we extend the value space so we can use the tag `FORWARD` to identify a forwarding pointer.[2] For debugging, it is also useful to add an `INVALID` tag; dead cells can be marked `INVALID`.

190a ⟨*value.t* 190a⟩≡

```
Lambda  = (Namelist formals, Exp body)
Value   = NIL
        | BOOL      (int)
        | NUM       (int)
        | SYM       (Name)
        | PAIR      (Value *car, Value *cdr)
        | CLOSURE   (Lambda lambda, Env env)
        | PRIMITIVE (int tag, Primitive *function)
        | FORWARD   (Value *)
        | INVALID   (const char *)
```

The basic operation used in a copying collector is *forwarding* a pointer. This operation takes a pointer to an object in from-space and returns a pointer to the unique copy of that object in to-space. The code is very simple; `p` is a pointer into from-space, and we copy the object `*p` unless it has been copied already. We copy the object to `*hp`, which is safe because while pointers are being forwarded, `hp` points into to-space, at the boundary between allocated and unallocated locations. Because to-space is as big as from-space, and because no object is copied more than once, there is guaranteed to be enough room to hold all the objects.

190b ⟨*forward pointer* `p` *and return the result* 190b⟩≡ (196b)

```
if (p->alt == FORWARD) {            /* forwarding pointer */
    return p->u.forward;
} else {
    *hp = *p;
    *p  = mkForward(hp);        /* overwrite *p with a new forwarding pointer */
    return hp++;
}
```

hp 194c
mkForward 𝒜
semispacesize 194b
tospace 194b

We can use the tricolor-marking idea to keep track of which pointers have been forwarded and which still need to be forwarded.

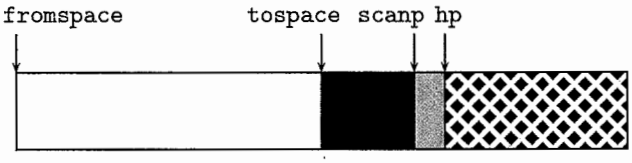
- An object is white if it is sitting in from-space. If it is reachable, it will need to be copied.

[1] In some systems, the copying component of the garbage collector has also been used as a marshaller.

[2] The `FORWARD` tag is not strictly necessary; we can identify a forwarding pointer simply by its value. A pointer is a forwarding pointer if and only if it points into to-space. For this trick to work (and indeed for copying collection to work at all), we must be able to distinguish pointers from non-pointers.

- An object is gray if it has been copied to to-space, but the pointers it contains have not been forwarded. (This implies that the objects it points to may not have been copied.)
- An object is black if it has been copied to to-space, and the pointers it contains have been forwarded. (This implies that the objects it points to have also been copied to to-space.)

We use an additional pointer, `scanp`, to mark the boundary between black objects and gray objects. White objects have addresses `fromspace` $\leq a <$ `fromspace` + `semispacesize`. Black objects have addresses `tospace` $\leq a <$ `scanp`. Gray objects have addresses `scanp` $\leq a <$ `hp`. Here is a picture:



Black objects point to to-space; gray objects point to from-space.

As always with tricolor marking, we begin with the roots, making the objects they point to gray. We then turn gray objects black until there aren't any more gray objects. In concrete terms, we first forward all the pointers that are in roots, then forward pointers in gray objects until there are no more.

191 ⟨*copy all reachable objects to to-space* 191⟩≡

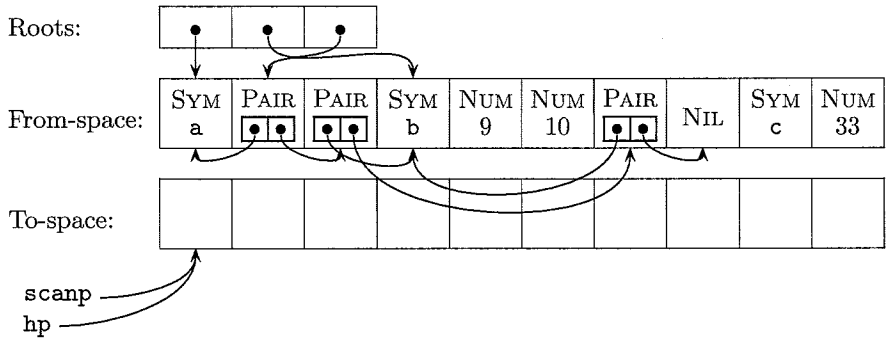
```
{
    int i;
    Value *scanp;

    scanp = hp = tospace;   /* no black or gray objects yet */
    for (i = 0; i < rootstacksize; i++) {
        ⟨scan root rootstack[i], forwarding all internal pointers 195b⟩
    }
    for (; scanp < hp; scanp++) {
        ⟨scan object *scanp, forwarding all internal pointers 195c⟩
    }
}
```

Further details are left for Exercise 12 and Appendix C.

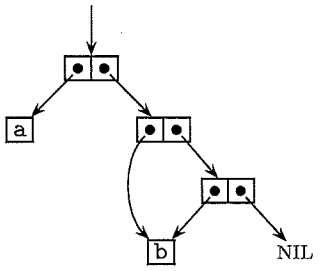
hp 194c
rootstacksize 182b
tospace 194b

4.5.2 A brief example

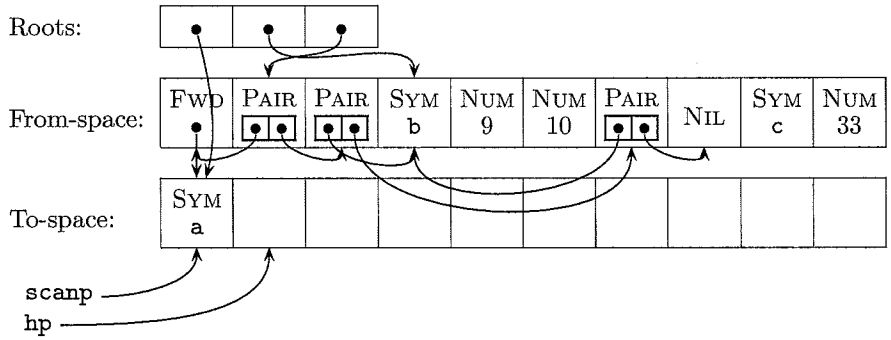
We illustrate copying collection using a heap of size 20. Suppose there are three roots, and the heap looks like this:



The reachable S-expressions happen all to be part of one list:

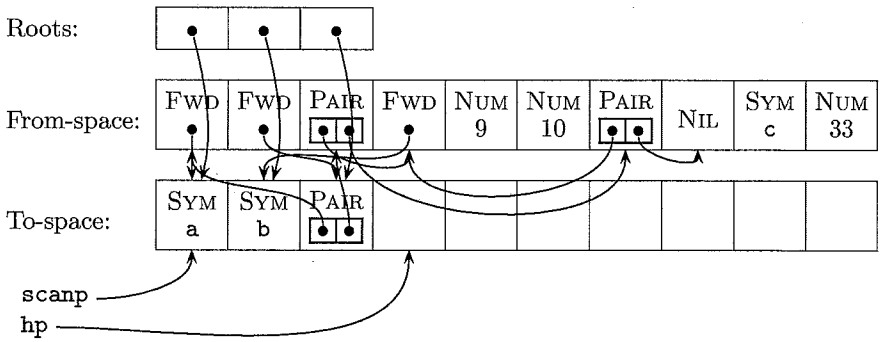


The first step in copying the live data is to forward the roots. After we forward the first root, the heap looks like this:



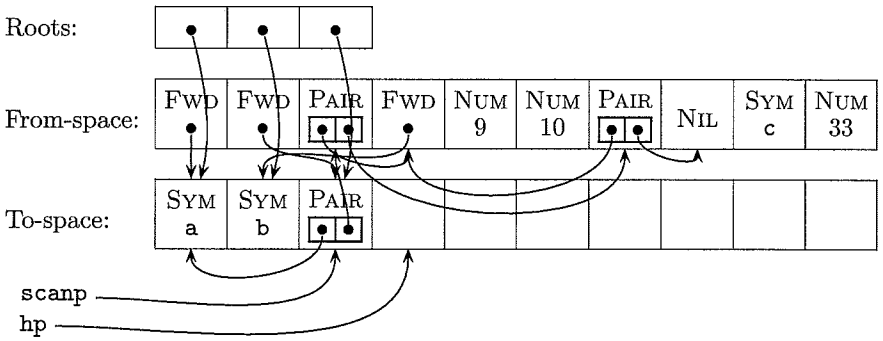
The first object in from-space has been copied to to-space and replaced with a forwarding pointer.

After all three roots have been forwarded, that is, after the execution of the first loop in ⟨*copy all reachable objects to to-space* 191⟩, the heap looks like this:

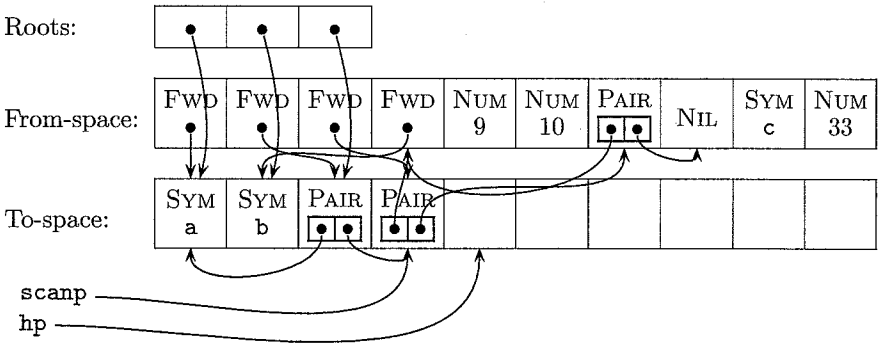


Now it is time to scan the gray objects located between `scanp` and `hp`. All the internal pointers in these objects point back to from-space, i.e., gray objects point to white objects.

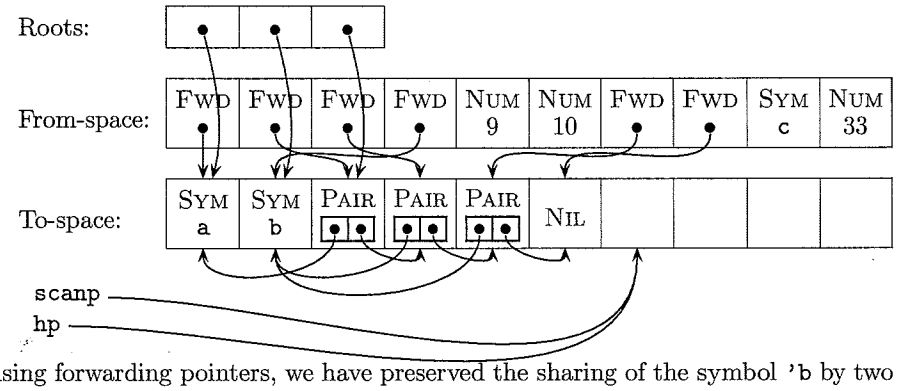
Scanning the first two gray objects does nothing, because they have no internal pointers. Scanning the third gray object (the pair) requires forwarding its two internal pointers. The symbol `'a` has *already* been copied to to-space, so forwarding the `car` doesn't copy any data; it just adjusts a pointer:



Forwarding the `cdr` requires copying another pair to to-space, however. After this copy, we adjust `scanp`, and the newly copied pair is now the only gray object—the first three objects in to-space are black.



We continue copying objects pointed to by `*scanp` until eventually `scanp` catches up with `hp`. Now there are only black objects in to-space and only white objects in from-space. From-space can be discarded, and we can continue execution, allocating from `hp` onward. We have recovered four locations to use to satisfy allocation requests.



By using forwarding pointers, we have preserved the sharing of the symbol `'b` by two pairs. The same technique also preserves cycles.

4.5.3 Prototype of a copying system for μScheme

Here we present more of the details of copying collection for μScheme. Further details appear in Section C.2, and to get a complete system, you can work Exercises 12 through 20.

194a ⟨*copy.c* 194a⟩≡ 194d▷
```
#include "all.h"
```

⟨*private declarations for copying collection* 194b⟩

The semispaces `fromspace` and `tospace` each have size `semispacesize`.

194b ⟨*private declarations for copying collection* 194b⟩≡ (194a) 194c▷
```
static Value *fromspace, *tospace;          /* used only at GC time */
static int semispacesize;                   /* size of fromspace and tospace */
```

We always allocate from `fromspace`. The heap pointer `hp` points to the next location available to be allocated, and `heaplimit` points just past the last location available to be allocated. The number of locations that can be allocated before the next collection is `heaplimit - hp`.

194c ⟨*private declarations for copying collection* 194b⟩+≡ (194a) ◁194b 195a▷
```
static Value *hp, *heaplimit;               /* used for every allocation */
```

collect 794a
mkInvalid A

Allocation

In a copying system, the allocator is very simple. In real systems, `hp` is in a register, and `allocloc` is inlined.

194d ⟨*copy.c* 194a⟩+≡ ◁194a 195d▷
```
Value* allocloc(void) {
    if (hp == heaplimit)
        collect();
    assert(hp < heaplimit);
    return hp++;
}
```

The assertion can help detect bugs in a heap-growth algorithm.

Tracing roots

Just as the mark-and-sweep system has a visiting procedure for each type of potential root, the copying system has a scanning procedure for each type of potential root.

195a ⟨*private declarations for copying collection* 194b⟩+≡ (194a) ◁194c 197a▷
```
static void scanvaluelist(Valuelist vl);
static void scanenv      (Env env);
static void scanexp      (Exp exp);
static void scanexplist  (Explist el);
static void scandef      (Def def);
static void scanvalue    (Value *vp);
static void scanroot     (Root r);
```

The function `scanroot` looks at the tag of a root and calls the appropriate scanning procedure.

Using these procedures, we can fill in the blanks in our copying routine:

195b ⟨*scan root* `rootstack[i]`, *forwarding all internal pointers* 195b⟩≡ (191)
```
scanroot(rootstack[i]);
```

195c ⟨*scan object* `*scanp`, *forwarding all internal pointers* 195c⟩≡ (191)
```
scanvalue(scanp);
```

The implementations of our scanning procedures are more complicated than they would be in real life. In a real system, these scanning procedures would simply forward internal pointers. In our system, because only `Value` objects are allocated on the heap, we have a hybrid of forwarding and graph traversal. As before, we use `Env` to illustrate; we forward the `loc` pointer but traverse the `tl` pointer.

195d ⟨*copy.c* 194a⟩+≡ ◁194d 196a▷
```
static void scanenv(Env env) {
    for (; env; env = env->tl)
        env->loc = forward(env->loc);
}
```

Here is the code to scan values. Again we forward pointers of type `Value*` but traverse other pointers.

196a ⟨*copy.c* 194a⟩+≡ ◁195d 196b▷

```
static void scanvalue(Value *vp) {
    switch (vp->alt) {
    case NIL:
    case BOOL:
    case NUM:
    case SYM:
        return;
    case PAIR:
        vp->u.pair.car = forward(vp->u.pair.car);
        vp->u.pair.cdr = forward(vp->u.pair.cdr);
        return;
    case CLOSURE:
        scanexp(vp->u.closure.lambda.body);
        scanenv(vp->u.closure.env);
        return;
    case PRIMITIVE:
        return;
    default:
        assert(0);
        return;
    }
}
```

The remaining scanning procedures appear in Section C.2.

Forwarding

The scanning procedures above are very similar to the visiting procedures in Sections 4.4.2 and C.1. One difference is that the `forward` operation, as shown in chunk ⟨*forward pointer* `p` *and return the result* 190b⟩, never makes a recursive call.

The complete implementation of `forward` has one more subtlety; in our system, the same root can appear on the root stack more than once.[3] If we scan such a root for a second time, its internal pointers already point to to-space. We need to test for this possibility so we don't try to forward a pointer that already points to to-space.

196b ⟨*copy.c* 194a⟩+≡ ◁196a

```
static Value* forward(Value *p) {
    if (isinspace(p, tospace)) { /* already in to space; must belong to scanned root */
        return p;
    } else {
        assert(isinspace(p, fromspace));
        ⟨forward pointer p and return the result 190b⟩
    }
}
```

fromspace 194b
scanexp 625
semispacesize 194b
tospace 194b

[3] This doesn't happen in real systems, because with compiler support, things naturally fall out such that each root is scanned *exactly* once. In our interpreter, without compiler support, it is much easier to ensure only that each root appears on the stack *at least* once.

We make `isinspace` a macro because measurements show it contributes significantly to garbage-collection time.

197a ⟨*private declarations for copying collection* 194b⟩+≡ (194a) ◁195a
```
#define isinspace(LOC, SPACE) ((SPACE) <= (LOC) && (LOC) < (SPACE) + semispacesize)
static Value *forward(Value *p);
```

A more aggressive collector would probably compute `isinspace` with a single comparison (Exercise 19).

Heap growth

In a mark-and-sweep collector, we can enlarge the heap simply by adding more arenas. In a copying system, semispaces have to be contiguous, and heap growth is more complicated. There's no way to extend something contiguously using `malloc` and `free`, but it is reasonable to proceed as follows:

1. Allocate a new, larger, to-space.
2. Copy the live data from from-space into to-space.
3. Flip from-space and to-space.
4. Allocate a new, larger, to-space.

A naïve implementation of this strategy would copy the live data twice every time the heap grows. A more sophisticated implementation delays the growth of the heap until the *next* collection (Exercise 16).

4.5.4 Performance

As for the mark-and-sweep system, we analyze the cost of a garbage-collection cycle with $N = \frac{H}{2} - L$ allocations, and we assume the heap is in steady state.

Allocation cost. In a copying system, the cost per allocation is a constant: test for heap exhaustion and increment the heap pointer. On modern RISC architectures, both the heap pointer and the `heaplimit` pointer are kept in registers. The allocator is just a few instructions, it is typically inlined, and it is very fast. Fast allocation is one of the principal advantages of a copying system.

For example, in a compiled system, we might implement `cons` this way:

197b ⟨*untested sample C code* 197b⟩≡
```
unsigned *hp, *heaplimit;

unsigned *cons(unsigned car, unsigned cdr) {
    if (hp+3 > heaplimit)
        gc();
    hp[0] = PAIR;
    hp[1] = car;
    hp[2] = cdr;
    hp += 3;
    return hp-3;
}
```

If the heap pointer, hp, is kept in a register, as is customary, this code requires a test, three stores, a move, and an add.

Copying allocators are especially good when we have objects of more than one size. At every allocation, a mark-and-sweep allocator must look at the size of the requests, then use multiple free lists, first fit, or some other strategy to find a chunk of free memory of an appropriate size. A copying allocator simply tests and increments.

Collector overhead. The total work done by a copying garbage collector is proportional to the size of the live data. The work itself is slightly more expensive than for the mark-and-sweep system, since we have to copy objects, not just set mark bits. The copying does produce a benefit, by compacting the live objects. Compaction improves the performance of the machine's cache and virtual memory, but more importantly, it enables fast allocation.

Memory overhead. A copying system has two sources of memory overhead:

- Because one semispace is always empty, the copying system requires $\gamma \geq 2$. The "cushion" needed to get adequate performance is much larger than in a mark-and-sweep system. As γ gets large, however, this difference between copying and mark-and-sweep disappears, and the cost of allocation dominates.
- Like the mark-and-sweep system, the copying system needs to know where to find pointers in heap objects.

The copying system has no marking overhead and requires no stack.

Pause time. Copying allocation always takes constant time; the only pauses are at collections. These pauses take time proportional to the amount of live data.

4.6 Debugging a garbage collector

There are two potential kinds of bugs in a garbage-collected system.

- *We fail to recycle some unreachable objects.* Such a *memory leak* does not affect correctness, only performance—and it may not be noticed for some time. Its only symptom is to cause some programs to run out of memory earlier than they should have, and it is usually hard to determine when they "should have."
- *We recycle an object that is still reachable.* When such an object is reused, its contents change. From the perspective of an old pointer to that object, the value changes for no reason, at an unpredictable time. Much time may pass before the object is reused, and even more time may pass before the bad value is detected. Such bugs, which could occur if we overlook a root or if our tracing procedures fail to follow a pointer, are extremely difficult to detect.

Memory leaks are not catastrophic bugs, and we say little about them. They can be detected by observing that the garbage collector is finding too much live data. Either regression testing or careful analysis may tell us how much is "too much." Once the problem is detected, one can examine the heap at collection time to find out exactly which objects are not being recycled and by what combination of pointers those objects are reachable.

Detecting premature recycling is a bit more difficult. Most of these kinds of bugs occur because the compiler has kept a root (a pointer to a heap object) but has neglected to tell the garbage collector. You are unlikely to have such problems while working the Exercises, because we provide tested root-tracking code in Section C.4. In other situations, the techniques below may help.

Memory-management bugs are unusually difficult to reproduce. Because garbage collections and allocations can happen at unpredictable times, bugs in the garbage collector are often only intermittently apparent. Worse, they can fail to manifest themselves until long after the initial fault. You can fight these problems by looking for faults early.

- Trigger a garbage collection before *every* allocation request. This trick may slow your program by many orders of magnitude, but you have a chance of detecting a missing root as soon as it disappears.
- After you run the collector, take a snapshot of the heap and run the collector again (once for mark-and-sweep, twice for copying). If the new heap is not identical to the snapshot, there is a bug in the collector. This technique will not detect a missing root.
- When your collector decides a location can be reused, mark the location as garbage in some way. (We use the special tag `INVALID`.) Change `eval` and `mkVL` to check the validity of *every* value they touch. If values like `'(a b c)` turn into `'(a b . <invalid>)`, you've discovered an early reclamation.
- Initialize newly allocated cells to some easily recognized value. We have used `mkInvalid("allocated but not initialized")`. If you have somehow failed to initialize a location and are following bad pointers, this technique may help.

For debugging a mark-and-sweep collector, try the following.

- If you are worried about memory corruption in your mark bits, use a whole word instead of a single bit. Instead of `0` and `1`, use a pair of unusual values like `0xdeadbeef` and `0xbadface`. If you ever see a different value, something is stomping on your mark bits.
- After marking the live objects, sweep the entire heap, marking non-live objects as invalid.

For debugging a copying collector, try the following.

- Initialize every value in a fresh to-space to `mkInvalid("never allocated")`.
- Before freeing an old space, set each of its values to `mkInvalid("dead")`, or perhaps `"dead`n`"`, where n is incremented for each space freed.
- Don't actually call `free` when you "free" old spaces. If you find values like `(dead . dead)`, you've found references to old spaces.
- Every time you copy live data into to-space, use the `mprotect` system call or an equivalent to make from-space inaccessible. Then allocate a new from-space before the flip. If you have mistakenly failed to forward some pointer, the next access through that pointer will cause a memory fault, which you can catch in a debugger.

4.7 Reference counting

Tracking roots appears to require either support from the compiler or run-time overhead.[4] If you can't get support from the compiler and you don't want to pay run-time overhead, you might look for other ideas. Because an object is definitely unreachable if there are no references to it, one idea is to keep track of the number of references to each object and to reuse an object when there are no references to it.

Of course, there may be more than one reference to an object. For example, in this session:

```
-> (set x '(a b))
-> (set y (car x))
```

there are two references to the `Value` object for `'a`. If `x` were reassigned, it would be wrong to deallocate that object. The effect of such premature deallocation would be that the object could be reused, causing `y` to obtain a new value rather mysteriously. To know for certain when there are no references to an object, we must keep track not only of *whether* there are references to a object, but of *how many* references there are. This number is called the object's *reference count*.

The idea of reference counting is to increment the reference count when a new reference is made to an object, decrement it when a reference is removed, and return the object to a free list when the reference count becomes zero. Here are some examples of where reference counts would be incremented and decremented in a reference-counting μScheme interpreter.

- A object has its reference count incremented when it is bound into to an environment, e.g., to be passed as a parameter or let-bound to a local variable.
- When a μScheme procedure exits, the interpreter decrements the reference counts of the procedure's formal parameters. It must do this correctly even on error exits.
- When the interpreter leaves the body of a let expression, it decrements the reference counts of let-bound objects.
- In `set`, the newly assigned value has its reference count incremented, and the previous value of the set variable has its reference count decremented.
- When a new cons cell is created, the reference counts of the car and cdr are incremented.
- When the reference count of a cons cell goes to zero, the reference counts of the car and cdr are decremented.

In short, reference counts need to be manipulated whenever pointers are manipulated. This requirement leads both to complexity in the compiler and to inefficiency in the compiled code. In a reference-counted system, *work per assignment* is as important as work per allocation; in a garbage-collected system, there is no per-assignment overhead. Although reference-counting overheads can be high, there need not be long pauses; it is easy to arrange things so that every allocation and assignment requires at most a constant amount of memory-management work.

[4] We write "appears to" because the ingenious technique developed by Boehm and Weiser (1988) works without either one.

Techniques exist for reducing reference-counting overhead, but garbage-collected systems can always run faster than reference-counting systems, since they get faster as available memory increases. There is no easy way to adjust overheads in a reference-counted system; the cost of each assignment and allocation is independent of the size of the heap. The flip side of this property is that reference counting easily outperforms garbage collection in systems with tight constraints on space, since reference counting performs no worse when γ is close to 1. Reference counting works perfectly well even when $H - L = 1$, where garbage collectors perform terribly.

Another problem with reference counting is that it cannot reclaim "cyclic garbage;" in a circular data structure, no count ever goes to zero. Primitives `set-car!` and `set-cdr!` can create circular structures. A naïve implementation of closures also leaves cyclic garbage: a closure contains an environment which contains a binding to a location containing the original closure. Various clever techniques have been proposed to help reference-counting deal with cyclic garbage, but they work only in limited domains. Reference counting seems most successful in languages that never create cycles.

Finally, although reference counting is conceptually simple, it can require pervasive changes to existing code, because *any* code that moves pointers has to be modified to adjust reference counts.

Reference counting seems to be most useful when used to do "double duty," i.e., when the reference counts are used not only to reclaim memory, but also to implement some other feature. One such feature is timely *finalization*, which is to perform some user-level task when an object is reclaimed. Such tasks may include closing open file descriptors or removing windows from a graphical user interface. Garbage-collected systems can also provide finalization, but it is seldom timely, because one can't predict how much time will elapse before a garbage collector reclaims a dead object.

Another feature for which reference counting can be useful is the *copy-on-write* optimization. This optimization applies to computations in which it is unsafe to mutate shared objects, so to play it safe, the program has to make a private copy, then mutate the copy. But if the reference count says there is only one reference to the object, then it isn't shared, and it is safe to mutate the original object without making a copy. This technique is used in the Newsqueak language to make array updates efficient in the common case (Pike 1990).

Reference counting also plays a starring role in distributed systems, where costs of local computation are almost irrelevant; what counts is minimizing the number of messages sent between computers. Following pointers, as is required for garbage collection, can require a lot of messages when the pointers cross machine boundaries. But the messages needed to adjust reference counts can often be combined with the messages used to refer to or assign values, so in terms of message traffic, reference counting comes essentially for free. Distributed architectures such as Java's Jini use reference counting for precisely this reason.

4.8 Garbage collection as it really is

In most garbage-collected systems, the compiler cooperates closely with the collector to help it identify roots and distinguish pointers from non-pointers.

- The compiler and collector agree on a calling convention for procedures, so the collector can find the activations of all the procedures on the call stack. For each source procedure, the compiler provides a map of the stack frame, which tells the collector where the roots are, i.e., which machine registers and stack variables point to heap objects. A good compiler is sophisticated enough to omit dead variables (i.e., variables the compiler knows will not be used again) from the stack map.
- The compiler identifies the global variables that point to heap objects.
- The compiler identifies the size and layout of each heap object, so the collector will know how big the object is what parts of it point to other heap objects. Methods vary widely. Most compilers put one or more *header words* before the heap object. A header word might point to a map that shows where the pointers are in the object, or it might point to a snippet of bytecode that the collector can interpret to find pointers within the object.

  In some systems, the header simply tells how many words there are in the object, and the compiler uses the low bit of each word as a marker to distinguish pointers from integers. Such systems are easy to identify, as they provide, e.g., only 31 bits of integer precision on a 32-bit machine.

To improve performance, state-of-the-art collectors use many variations on the basic collectors discussed above.

- Most objects die young; if an object survives even one garbage collection, it is likely to live a long time. *Generational* collectors exploit this property to reduce garbage-collection overheads. They divide the heap into two or more *generations*. Objects are allocated in the youngest generation, or *nursery*, which is collected most frequently. When the nursery fills up, it is collected, and objects that survive the collection are *promoted* into an older generation. Many collections of the nursery, or *minor collections*, are possible before a *major collection* is required to collect an older generation.

  A generational collector can let γ be large for the nursery but smaller for older generations. This arrangement achieves most of the performance benefit of large γ at a fraction of the memory cost. Some general systems use a hybrid collector; objects are copied out of the nursery, keeping free space contiguous, but older generations are collected using mark-and-sweep techniques. Using this trick, we can cut γ in half for the older generations.

  Another advantage of a nursery is that in a copying collector, a nursery can help allocation run even faster. If a collector allocates an entire semispace between collections, allocation is likely to cause a cache miss every time the heap pointer reaches a new cache line. If allocation takes place only in a nursery, we can arrange for the nursery to be small enough that it always fits in the cache. Voilà! No more cache misses at allocations.

A key issue in generational collection is identifying pointers from old objects to new objects. Such a pointer must be treated as a root for purposes of minor collection. Such a pointer can arise only when a program mutates the contents of a previously allocated object. In a generational system, such a mutation carries additional overhead.[5] Fortunately, such mutations are impossible in μScheme, and they are rare in functional languages such as Scheme and ML.

- It is possible to reduce the number of header words, which describe the sizes and internal layouts of objects, by reserving entire pages of memory to hold only objects of the same type. This strategy, called Big Bag of Pages, or BIBOP, uses a single header word for an entire page of memory. This strategy reduces memory overhead, but it makes it more expensive to find the header word associated with a particular object.

- Many systems, especially those based on copying collectors, use separate areas of memory to hold large objects. These large-object areas are treated specially, and they are not copied, even by a copying collector. This specialized strategy is especially useful for storing large strings or bitmaps, which do not contain pointers and so need not even be scanned. Such objects can be added to the live data of an advanced system without increasing garbage-collection times proportionally.

- Modern collectors can work even without support from the compiler. The collector developed by Boehm and Weiser (1988) is a mark-and-sweep collector that works without any compiler support at all. It works by searching the *entire* call stack for roots, as well as the program segments that hold initialized and uninitialized data. If any word anywhere in those areas appears to point to a heap-allocated object, that object is assumed to be live. (Because it is in cahoots with the allocator, the garbage collector knows the addresses of all the heap-allocated objects.) Similarly, if any word anywhere in a heap-allocated object appears to point to another heap-allocated object, the second object is also assumed to be live. This collector is called *conservative* because it conservatively assumes that any word might be a pointer. It has been used widely to achieve remarkable results with C and C++ programs, sometimes outperforming `malloc` and `free`.

- With just a little compiler support, it is possible to make a conservative copying collector, or rather a *mostly-copying* collector. This idea was developed for a system in which the compiler knows the layout of heap-allocated objects, but the call stack is a mystery. This situation arises whenever a high-level language is implemented by compiling to C code.

  A mostly-copying collector works by first scanning the stack for words that point into heap-allocated objects. Any heap objects pointed to by one of these words are *pinned* and cannot move. But heap objects that are pointed to only by other heap objects can be moved with impunity, since the compiler tells the collector exactly where the internal pointers are. The mostly-copying collector can achieve many of the performance benefits of a full copying collector (Bartlett 1988).

- By interleaving very short bursts of garbage collection with real work, it is possible to reduce pause times to 60msec or less. On multiprocessors, it is possible to run the garbage collector on its own processor, sometimes eliminating pauses altogether. Both these techniques are difficult to implement.

[5] The overhead of mutation is not the same as the overhead of assignment in a reference-counted system. In a reference-counted system, even an assignment to a local or global variable may carry run-time overhead.

4.9 Summary

No memory-management technique can help when the legitimate memory needs of a program exceed the memory available. The techniques of this chapter, however, can be used to limit the use of memory to just those legitimate needs, avoiding spurious memory exhaustion.

The key to understanding the performance of a garbage-collected system is to remember that *the collector and allocator work as a team*, and they must be analyzed as a team. The most important metric for most programs is *work per allocation*.

In situations where allocation rates are fairly low, so the cost of allocation is not a bottleneck, mark-and-sweep systems perform well, without requiring much more memory than is needed to hold the live data. It is also fairly easy to migrate existing `malloc/free` code to use a mark-and-sweep system; the garbage collector simply decides when to call `free`.

A copying system has two key properties: allocation is very fast, and *objects in the heap move*. These properties are, of course, related; we make allocation fast by making free space contiguous, and we make free space contiguous by moving objects. This *compaction*—making both live data and free space contiguous—eliminates fragmentation, enables fast allocation, and improves locality. When allocation rates are high, copying systems perform very well, because allocation is so fast. Copying collection can be more difficult to implement, however; because compaction moves objects, the compiler must be aware that objects move, and we must know exactly where all the pointers are.

In both copying and mark-and-sweep systems, we can reduce garbage-collection overhead by increasing the size of the heap. When we can afford to make γ large, allocation cost dominates, and copying collection usually wins.

The real cost of automatic memory management depends not just on the collection technique but on the program. Simple programs that don't allocate much may spend only a few percent of their time in garbage collection. Programs that allocate heavily may spend up to 40 or 50 percent of their time in garbage collection. Comparisons with explicit memory management reveal that garbage collection can be *faster* than `malloc` and `free`, even on programs that aren't designed to exploit a garbage collector (Zorn 1993). The price paid is that a garbage-collected system needs more memory, perhaps up to twice as much.

Since the time to perform any operation that requires memory allocation—`cons` in Scheme, `new` in Smalltalk, etc.—must be calculated as the sum of allocation time, operation time, and its share of garbage-collection time, these operations are generally viewed as expensive. Some programmers go to great lengths to minimize such operations, e.g., by using `set-car!` and `set-cdr!` to avoid `cons`. Such tricks tend to render code obscure and brittle, and developments in the 1990s have rendered them largely unnecessary. While it is never wise to allocate gratuitously, today's garbage-collected systems perform well enough that programmers need not use mutation merely for performance reasons. In fact, mutation makes some modern generational collectors perform *worse* than they would otherwise.

4.9.1 Glossary

compacting garbage collection Any method of garbage collection in which live objects are moved to contiguous memory locations. By making free space contiguous, compacting collectors can greatly speed allocation. On systems that use a cache or virtual memory, compacting collectors can also improve performance by increasing locality.

garbage collection An approach to memory management in which unreachable, or garbage, cells are recovered not at the instant they become unreachable but rather in a separate phase or thread of computation.

heap-based memory allocation When the order of allocation of chunks of memory cannot be related to the order of their *de*-allocation, they must be allocated on a heap. Memory-management techniques such as reference counting and garbage collection can relieve the programmer of the obligation to know when it is safe to deallocate such objects.

mark-and-sweep garbage collection In this method, the active objects in memory are marked during a recursive traversal. This traversal is followed by a sweep of all of memory, which reconstructs the free list from the unmarked cells. The method is also known as **mark-scan** collection.

reference counting A memory-management technique in which each chunk of allocated memory contains a *reference count*, which holds the number of references to that chunk. The reference count is incremented and decremented as references are added and removed, and the memory can be recovered at the instant the reference count goes to zero.

copying garbage collection In this method, live objects are copied from one place to another; the copying has the effect of compacting as well. The simplest copying collectors use only half of memory at a time. When one half is exhausted, the collector copies all live data to the beginning of the other half. This method is also called **stop-and-copy** collection.

stack-based memory allocation When the last chunk of memory to be allocated is always the first to be de-allocated, memory can be allocated from a stack. The standard example is the allocation of memory for the formal parameters and local variables of a procedure or function, whether in our interpreters or in conventional languages like C.

4.9.2 Further Reading

The books by Knuth (1973) and Standish (1980) go into detail on a variety of memory management techniques; the survey paper by Cohen (1981) is also excellent. Jones and Lins (1996) present a comprehensive survey of more recent techniques for automatic memory management. The survey by Wilson (1992) is less comprehensive, but it thoroughly covers garbage-collection techniques for uniprocessors. All have extensive lists of references.

The mark-and-sweep method was outlined by McCarthy (1960); the technique of marking nodes without an additional stack is credited to Deutsch and independently to Schorr and Waite (1967); the books by Knuth and Standish have more information. Minsky (1963) describes a compacting collector that copies live data to tape, then back into main memory. Minsky points out that a copy phase has the added benefit of linearizing `cdr` chains and compacting both the entire heap and individual objects (like lists). Fenichel and Yochelson (1969) describe in-memory copying collection; Baker (1978) developed a well-known variant, which avoids the need to stop the computation to do garbage collection. Lieberman and Hewitt (1983) describe real-time collection, and Ungar (1984) introduced generational collection. Appel (1989) describes generational collection in a very clear, simple setting.

Boehm and Weiser (1988) introduced the idea of *conservative* collection; Boehm and several of his colleagues have developed this conservative collector into a very sophisticated component that can easily be added to almost any C program. Zorn (1993) reports on experiments with the Boehm collector, showing that even without compiler support, garbage collection is competetive with `malloc` and `free`, and sometimes faster. Bartlett (1988) shows how conservative techniques can be used even in a copying system. Smith and Morrisett (1999) compare the two styles of conservative collection.

As of Spring 2002, tools for analyzing and understanding the performance of managed heaps are not widely deployed. Runciman and Wakeling (1993) describe one of the first experiments with "heap profiling." Serrano and Boehm (2000) describe other tools used to understand the performance of a managed heap.

Stack-based memory is generally thought to be far more efficient than heap memory, because locations can be reclaimed from a stack simply by adjusting the stack pointer, but Appel (1987) presents an interesting dissenting argument.

4.10 Exercises

Learning about mark-and-sweep garbage collection

Exercises 1–11 involve building and analyzing a mark-and-sweep system for μScheme. The code in Appendix C should be helpful.

1. Write a procedure `mark` that implements the mark phase of a mark-and-sweep garbage collector. At each collection, traverse the root set and mark each reachable `Mvalue` as live. Use the visiting procedures in Sections 4.4.2 and C.1. Your `mark` procedure should call the appropriate visiting procedure for each of the roots.

2. After completing Exercise 1, instrument your collector to gather statistics about memory usage and live data in the μScheme heap. (The μScheme heap, i.e., the collection of all arenas, is not the C heap. The μScheme heap is managed with `allocloc` and `mark`; the C heap is managed with `malloc` and `free`.)

   (a) After every collection, have the `mark` procedure print out the total number of locations on the μScheme heap (the heap size), the number that hold live objects at the given collection (the live data), and the ratio of the heap size to live data. To gather this information, you will need to record the total number of mark bits set in the `mark` phase.

   (b) Gather and print statistics about total memory usage; print the total number of cells allocated and the current size of the heap after every 10th garbage collection and when your interpreter exits. This information will give you some intuition on how garbage collection allows you to reuse memory; with garbage collection, some of the programs in this book can run in 60 times less memory than without garbage collection.

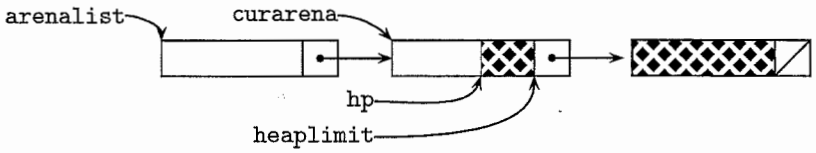
   (c) When your interpreter exits, print out the total number of collections and the number of cells marked during those collections.

3. Create an allocator that, together with your `mark` procedure from Exercise 2, forms a complete mark-and-sweep system.

   - The allocator must implement not only allocation but also the unmark and sweep phases of collection. Such an allocator works by sweeping through the heap from the first arena to the last arena. Instead of just taking the cell pointed to by `hp`, as in chunk 185c, the allocator must check to see if it is marked `live`. If marked, the location was live at the last collection, so it cannot be used to satisfy the allocation request. Skip past it and mark it not live. This change effectively moves both unmark and sweep phases into the allocator.

     When the allocator finds an unmarked object, it sweeps past the object and returns it to satisfy the allocation request, just as in chunk 185c. If the allocator gets to the end of the heap without finding an unmarked object, it calls `mark`.

   - In the new system, end-of-arena is not the same as end-of-heap. For example, after some number of allocation requests, the heap might look like this:



     White areas, to the left of the heap pointer, have been used to satisfy previous allocation requests. Areas marked with black diamonds, to the right of the heap pointer, are potentially available to satisfy future allocation requests.[6] The code in chunk 185c must be modified to look in the next arena when the current arena is exhausted. Only after there are no more arenas should it call `mark`.

   - After your allocator calls `mark`, it should call `makecurrent` to reset `hp`, `heaplimit`, and `curarena` to point into the first arena.

   - The allocator sketched here guarantees that when `mark` is called, the entire heap is unmarked (see Exercise 4). However, the `mark` procedure is not guaranteed to recover any garbage; every cell on the heap might be live. The allocator may have to enlarge the heap by adding a new arena. Write a procedure `growheap` for this purpose.

   You may find it helpful to split `allocloc` into two functions: one that attempts to allocate without calling `mark`, but sometimes fails, and another that may call the first function, `mark`, and `growheap`.

   You are likely to have an easier time with your implementation if you work on Exercise 4 first. A thorough understanding of the invariants should make the implementation relatively easy.

4. Prove that in the scheme outlined in Exercise 3, `mark` is called only when no S-expression is marked `live`.

[6] They are only "potentially" available because if they are marked, they were live at the last collection and must be assumed to be live now. The allocator must unmark and skip over such cells.

Here is a sketch of one possible proof.

(a) Find an invariant for the current arena pointed to by `curarena`. You'll need one property for `curarena->pool[i]` when `&curarena->pool[i] < hp`, and a different property for `curarena->pool[i]` when `&curarena->pool[i] >= hp`. Your `growheap` procedure will have to establish the invariant for every new arena.

(b) Extend the invariant to include arenas before and after the current `curarena`. The invariants for these two kinds of arenas should look an awful lot like the invariants for `curarena->pool[i]` where `&curarena->pool[i] < hp` and `&curarena->pool[i] >= hp`. Together, these invariants describe the white and black-diamond areas in the picture of the heap.

(c) Show that `growheap`, `allocloc`, and `mark` all maintain this invariant.

(d) Show that conditions when `mark` is called, together with the invariant, imply that `mark` is called only when no S-expression is marked `live`.

5. The performance of a garbage collector depends on the size of the heap, which should be controllable by the programmer. The function `gammadesired`, which is defined in chunk 627b, makes it possible to use the μScheme variable `&gamma-desired` to control the size of the heap. Because μScheme doesn't have floating-point support, the integer `&gamma-desired` represents 100 times the desired ratio of heap size to live data.

   (a) Modify your collector so that after each collection it enlarges the heap, possibly by adding more than one arena, until the measured γ is as close as possible to $\frac{1}{100}$`&gamma-desired`, without going under. For example, executing `(set &gamma-desired 175)` should cause your collector to increase the heap size to make γ about 1.75. (Do not try to make the heap smaller if γ is too big; see Exercise 8.)

   (b) Measure the amount of work done *by the collector* for different values of `&gamma-desired` and for different programs. Think carefully about the *units* in which work is measured.

   (c) Plot a graph that shows *collector work per allocation* as a function of the value of `&gamma-desired`.

   (d) Using your measurements from part (b), choose a sensible default value for γ in case the programmer has not set `&gamma-desired`.

6. Derive a formula to express the cost of mark-and-sweep garbage collection as a function of γ. The cost should be measured in units of *GC work per allocation*. For a very simple approximation, assume a fixed percentage of whatever is allocated becomes garbage by the next collection.

   You should not find it necessary to assume that you know anything else about the heap besides γ.

7. If you have done Exercises 5 and 6, discuss how your formula compares with your measurements. Explain any inconsistencies.

8. Because objects in our mark-and-sweep collector don't move, we can shrink the heap only when we find a completely empty arena. Assume that $\gamma = 10$, so the heap is 90% empty, and that `GROWTH_UNIT` is chosen so that one arena just fits on a virtual-memory page (about 4KB to 8KB). What is the probability that an arena chosen at random has no live objects in it and can be removed from the heap?

9. The mark-and-sweep allocator skips marked cells until it finds an unmarked cell. Assume that the marked cells are distributed independently, so that the probability of any particular cell being marked is $\frac{1}{\gamma}$.

   (a) Derive a formula giving, as a function of γ, the expected number of marked cells the allocator must skip before finding an unmarked cell. Calculate this number for some interesting values of γ, e.g., $\gamma \in \{1, 1.1, 1.2, 1.5, 2, 3, 5\}$.

   (b) Derive a formula giving, as a function of γ, the probability that the allocator skips at most 3 cells. Calculate this probability for some interesting values of γ.

10. Our system puts mark bits in the object headers even though there's no spare bit and we therefore have to allocate an extra word per object.

    (a) Rewrite the allocator and collector to pack the mark bits into one or more words in the arena. You will need to be able to map the address of an object to the address of the arena containing that object. If you arrange for the addresses of arenas to fall on k-bit boundaries, where `GROWTH_UNIT` $< 2^k$, you can find the address of the arena simply by masking out the least significant k bits of the address of the object.

    (b) Using the new representation, how many more objects can fit into an arena of the same size? What does this result imply about choosing γ?

11. Instead of using an explicit root stack, look at the actual C stack and conservatively estimate that every word on the C stack could be a pointer into the heap. Use this estimate to mark all the heap objects that might possibly be live.

    You can find the bounds of the C stack by looking at the address of a local variable in `main` and the address of a local variable in `allocloc`.

    You will need to write the following function:

209 ⟨*untested prototypes for exercises* 209⟩≡
```
Mvalue *whatobject(void *p);
```

    If `p` points into a heap object, `whatobject` returns the address of that object. Otherwise, `whatobject` returns `NULL`. The bit-masking technique described in Exercise 10 is useful; Boehm and Weiser (1988) give more details.

Learning about copying collection

Exercises 12–20 involve building and analyzing a copying system for μScheme. The code in Appendix C should be helpful.

12. Write a `copy` procedure that copies all live objects to to-space and leaves `hp` and `heaplimit` pointing to appropriate locations in to-space.

13. Write a function `collect` to be called from `allocloc` in chunk 194d. This function should call `copy`, perform the flip, and possibly enlarge the heap.

14. Instrument your collector to gather statistics about memory usage and live data in the μScheme heap, much as in Exercise 2.

    (a) Have your `copy` procedure print out the total number of locations on the heap, the number that hold live objects at the given collection, and the ratio of the heap size to the amount of live data.

    (b) Gather and print statistics about total memory usage; print the total number of cells allocated and the current size of the heap after every 10th garbage collection and when your interpreter exits.

    (c) When your interpreter exits, print out the total number of collections and the number of objects copied during those collections.

15. Using `&gamma-desired`, as described in Exercise 5, modify `collect` so that after each collection it enlarges the heap until the actual γ is at least $\frac{1}{100}$`&gamma-desired`.

    (a) Measure the amount of work done *by the collector* for different values of `&gamma-desired` and for different programs. Think carefully about the *units* in which work is measured.

    (b) Plot a graph that shows *collector work per allocation* as a function of the value of `&gamma-desired`.

    (c) Using your measurements from part (a), choose a sensible default value for γ in case the programmer has not set `&gamma-desired`.

    Remember that γ is the ratio of the *total heap size* to the amount of live data, *not* the ratio of the size of a semi-space to the amount of live data.

16. Enlarging the heap immediately after a `copy` operation requires a *second* `copy` operation to get the live data into the new heap. The work of this second `copy` is completely wasted, as no allocations can be charged against it. Change your implementation of `collect` so that if the heap needs to be enlarged, `collect` delays the job until just before the *next* collection. Measure the difference in GC work per allocation for both short-running and long-running programs.

17. Derive a formula to express the cost of copying garbage collection as a function of γ. The cost should be measured in units of *GC work per allocation*. The units of work may be different from the units you used in Exercise 6. For a very simple approximation, assume a fixed percentage of whatever is allocated becomes garbage by the next collection.

    You should not find it necessary to assume that you know anything else about the heap besides γ.

18. If you have done Exercises 15 and 17, discuss how your formula compares with your measurements. Explain any inconsistencies.

19. Chunk 197a defines a version of `isinspace` that uses two comparisons. Assuming that pointer comparison can be implemented with unsigned arithmetic, rewrite the macro `isinspace` to use one subtraction and one unsigned comparison. Measure the effect of this rewrite on the speed of garbage collection.

20. Read the paper by Appel (1989) and modify your copying collector so it uses two generations. In the new system, measure work per allocation as a function of γ.

Learning about other variations

21. Eliminate `Valuelist` objects altogether by passing arguments on a stack. The values on this argument stack should be roots for garbage collection. Functions `eval` and `evallist` should push their results onto the stack, and `bindalloc` and `bindalloclist` should take their arguments from the stack.

    Measure the change in the amount of memory consumed by the μScheme interpreter.

22. Pick a collector and measure the distributions of objects' lifetimes for several programs. Measure lifetimes in units of *number of objects allocated.* You will need to add a field to each object that tells when it was allocated, and at each collection you will be able to estimate the lifetimes of the objects that died at that collection. The accuracy of your estimate will depend on the number of allocations that take place between collections.

    Does the distribution of lifetimes depend on γ? Why or why not?

23. Some time may elapse between when an object is last used and when it becomes unreachable. This time, called the *drag time* of the object, contributes to excess space usage in garbage-collected systems. Drag time should be measured in units of *number of objects allocated.* Add instrumentation to the interpreter to measure drag times:

    (a) Add a field to each object that tells when it was last used. Every use of this object should update this field. A reasonable approximation is to write the current time into the field every time an object is looked up in an environment or obtained by `car` or `cdr` operations.

    (b) Add another field to each object that tells when it was last known to be reachable. This field can be updated by the garbage collector, or for more accurate measurements, you can run a periodic check to update these fields. The code is just like the code for `mark`.

    (c) When an unreachable object is garbage-collected, log the two times (time of last use and time of unreachability) to a file. When the program ends, log the same information for the remaining objects.

    (d) Write a program to read the log and compute two curves. The *in-use curve* records the number of objects in use as a function of time. The *reachable curve* records the number of objects that are reachable, also as a function of time.

    (e) The area under each curve is a good measure of space usage. The ratio of in-use space to reachable space is a good measure of the degree to which reachability overestimates the lifetimes of objects. Compute this ratio for several executions of several programs.

24. Although traditional copying collection moves all free space into one contiguous region of memory, it is also possible to write a copying collector in which the heap is divided into arenas (Bartlett 1988). Such a system has several advantages:

    - Because heap space need not be contiguous, the garbage-collected heap can more easily coexist with memory that is managed using other techniques, such as explicit `malloc` and `free`.
    - If for some reason it is not safe to move an object (perhaps because a device driver is reading its contents), that object can be "pinned" while still allowing copying collection of other objects.
    - The heap can be enlarged simply by adding more arenas; the collector does not need the sort of delaying tactics described in Exercise 16.

    Modify your copying collector to use arenas.

    (a) Develop data structures to represent each semispace as a list of arenas.

    (b) Do not change any code in `allocloc`. Instead, develop a new invariant to explain the meaning of the pointers `hp` and `hplimit`.

    (c) Change `collect` so that in the common case, it simply starts allocating from the next free arena. Only if all free arenas have been exhausted should you copy all reachable objects to to-space.

    (d) Change your heap-enlargement code to add arenas, not to allocate large, contiguous semispaces.

    Measure or estimate some of the cost/benefit tradeoffs:

    (e) For a program with lots of heap growth, measure the *collector work per allocation* and compare with the amount of work you measured in Exercise 15.

    (f) Estimate the number of machine instructions required to allocate an object in the common case, and estimate the number of machine instructions required to start allocating from the next free arena. Given your estimates and the number of objects per arena, what is the percentage overhead that arena-based allocation adds to the original allocator in chunk 194d? In a production system, you might allocate three-word cons cells into 4KB arenas; what overhead would you expect then?

25. Our implementation of `lambda` captures the entire current environment, including all live variables. As long as the closure is live, all those variables and the heap objects they point to, transitively, are live. But to implement `lambda` the only variables we really need in the closure are variables that appear free in the body of the `lambda`. The other variables may consume space unnecessarily.

    (a) Re-implement `lambda` to build a new environment containing only the variables that are actually needed. Function `freevars` from Appendix B may be useful.

    (b) Measure how this change affects the performance of the garbage-collected heap. As your measure of performance, you may choose to use number of collections, equilibrium heap size, average lifetime of objects, or drag time. Drag time is probably the most interesting measure, since our implementation "drags" these unneeded variables along for the ride.