How does your favorite object-oriented library solve this problem?

Mixins do have one limitation: they enforce a linearity of composition. This strictness is sometimes misplaced, because it puts a burden on programmers that may not be necessary. A generalization of mixins called *traits* says that instead of extending a single mixin, we can extend a *set* of them. Of course, the moment we extend more than one, we must again contend with potential name-clashes. Thus traits must be equipped with mechanisms for resolving name clashes, often in the form of some name-combination algebra. Traits thus offer a nice complement to mixins, enabling programmers to choose the mechanism that best fits their needs. As a result, Racket provides both mixins and traits.

11 Memory Management

11.1 Garbage

We use the term *garbage* to refer to allocated memory that is no longer necessary. There are two distinct kinds of allocations that a typical programming language run-time system performs. One kind is for the environment; this follows a push-and-pop discipline consistent with the nature of static scope. Returning from a procedure returns that procedure's allocated environment space for subsequent use, seemingly free of cost. In contrast, allocation on the store has to follow an value's lifetime, which could outlive that of the scope in which it was created—indeed, it may live forever. Therefore, we need a different strategy for recovering space consumed by store-allocated garbage.

It's not free! The machine has to execute an explicit "pop" instruction to recover that space. As a result, it is not *necessarily* cheaper than other memory management strategies.

There are many methods for recovering this space, but they largely fall into two camps: manual and automatic. Manual collection depends on the developer being able to know and correctly discard unwated memory. Traditionally, humans have not proven especially good at this (though in some cases they have knowledge a machine might not [REF]). Over several decades, therefore, automated methods have become nearly ubiquitous.

11.2 What is "Correct" Garbage Recovery?

Garbage recovery should neither recover space too early (*soundness*) nor too late (*completeness*). While both can be regarded as flaws, they are not symmetric in their impact: arguably, recovering too early is much worse. That is because if we recover a store location prematurely, the computation will continue to use it and potentially write other data into it, thereby working with nonsensical data. This leads at the very least to program incorrectness, but in more extreme situations leads to much worse phenomena such as security violations. In contrast, holding on to memory for too long decreases performance and eventually causes the program to terminate even though, in a platonic sense, it had memory available. This performance degradation and premature termination is always annoying, and in certain mission-critical systems can be deeply problematic, but at least the program does not compute nonsense.

Ideally we would love to have all three: automation, soundness, and completeness. However, we face a classic "pick two" tradeoff. Ideal humans are capable of attaining both soundness and completeness, but in practice rarely achieve either. A computer can offer automation and either soundness or completeness, but computability arguments demonstrate that automation can't be accompanied by both of the others. In practice, therefore, automated techniques offer soundness, on the grounds that: (a) it does the least harm, (b) it is relatively easy to implement, and (c) with some human intervention it can more closely approximate completeness.

You, surely, are perfect, but what of your fellow developers? And by the way, the economics discipline has been looking for you.

11.3 Manual Reclamation

The most manual approach would be to entrust all de-allocation to the human. In C, for instance, there are two basic primitives: `malloc` for allocation and `free` to reclaim. `malloc` consumes a size and returns a reference to a store-allocated value; `free` consumes the references and reclaims its assocated memory.

"Moloch has been used figuratively in English literature from John Milton's *Paradise Lost* (1667) to Allen Ginsberg's 'Howl' (1955), to refer to a person or thing demanding or requiring a very costly sacrifice." —Wikipedia on Moloch

"I do not consider it coincidental that this name sounds like `malloc`." —Ian Barland

11.3.1 The Cost of Fully-Manual Reclamation

Let's start by asking what the cost of these operations is. We might begin by assuming that `malloc` has an associated register pointing into the store (like `new-loc` [REF]), and on every allocation it simply obtains the next free locations. This model is extremely simple—in fact, deceptively so. The problem arises when we `free` these values. Provided the first `free` is the last `malloc`, we would encounter no problem; but store data often do not follow a stack discipline. When we free anything but the most recently allocated value, we leave holes in the store. These holes lead to *fragmentation*, and in the worst case we become unable to allocate any objects even though there is ample space in the store—just split up across many fragments, no one of which is large enough.

Exercise

> In principle, we could eliminate fragmentation by making all the free space be contiguous. What does it take to do so? Think through all the consequences and sketch whether you can in fact do this manually.

While fragmentation remains an insuperable problem in most manual memory management schemes, there is more to consider even in this seemingly simple discipline. What happens when we `free` a value? The run-time system has to somehow record that it is now available for future allocation. It does by maintaining a *free list*: a linked-list of the free spaces. A little reflection immediately suggests a question: where is the free list itself stored, and who manages *its* memory? The answer is that the free list references are stored in the freed cells, which immediately implies a minimum size for each allocation.

In principle, then, each `malloc` must now traverse the free list to find a suitable freed spot. I say "suitable" because the allocator must make a complex decision. Should it take the first slot that matches or a later one? And either way, what does "matches" mean? Should it take only slots the exact right size, or take larger slots and break them up into smaller ones (thereby increasing the likelihood of creating unusably small holes)? And so on.

Developers like allocation to be cheap. Therefore, in practice, allocation systems

Failing to make allocation cheap makes developers try to encode tricks based on reusing values, thereby reducing clarity and quite possibly also correctness.

tend to use just a fixed set of sizes, often in powers of two. This makes it possible to maintain not one but many free list*s*, each of holes of the same size (which is a power of two). A table refers to each of these lists, and indexing in the table is cheap by using bit-shifting. In return, developers sacrifice space, because objects not a power-of-two size will end up being needlessly padded. (This is a classic computer science trade-off: trading space for time.) Also, `free` must put the freed memory in the right slot, and perhaps even break up larger blocks into multiple smaller blocks to prepare for future allocations. Nothing about this model is inherently as cheap as it seems.

In particular, `free` is not free.

Of course, all this assumes that developers can function in even a sound, much less complete, fashion. But they don't.

11.3.2 Reference Counting

Because entirely manual reclamation puts an undue burden on developers, some semi-automated techniques have seen long-standing use, most notably *reference counting*.

In reference counting, every value has associated with it a count of how many references it has. The developer is responsible for incrementing and decrementing these counts. When the count reaches zero, the value's space can safely be restored for future reuse.

Observe, immediately, that there are two significant assumptions lurking under this simple definition.

1. That the developer can track every reference. Recall that every alias is also a reference. Thus, a developer who writes

   ```
   (let ([x <some value>])
     (let ([y x])
       ...))
   ```

   has to remember that `y` is a second reference to the same value being referred to by `x`, and increment the count accordingly.

2. That every value has only a finite number of references. This assumption fails when a value has cycles.

Because of the need to manually increment and decrement references, this technique suffers from a lack of both soundness and completeness. Indeed, the second assumption above naturally leads to lack of completeness, while the first assumption points to the simplest way to break soundness.

The perils of manual memory management are subtle and run deeper. Because developers are charged with freeing memory (or, equivalently, managing reference counts), the policy of memory management has to become part of every library's interface: effectively, "Who's going to de-allocate values allocated by this library, and will the library de-allocate values passed to it?" It is unfortunately difficult to document and follow these policies precisely, but even worse, it pollutes the description of the library with low-level details that usually have nothing to do with its intended behavior.

One intriguing idea is to *automate* the insertion of reference increments and decrements. Another is to add cycle-detection to the implementation. Doing both solves many of the above problems, but reference counting suffers from others, too:

- The reference count increases the size of each object. It has to be large enough to not overflow, yet small enough to not appreciably increase the program's memory footprint.
- The time spent to increase and decrease these counts can become significant.
- If an object's reference count becomes zero, everything it refers to must also have their reference counts decreased—possibly recursively. This means a single deallocation action can have a large time impact, barring clever "lazy" techniques (which then increase the program's memory footprint).
- To decrement the reference count, we have to walk objects that are *garbage*. This seems highly counterproductive: to traverse objects we are *no longer interested in*. This has practical consequences: objects we are not interested in may not have been accessed in a while, which means they might have been paged out. The reference counter has to page them back in, just to inform them that they are no longer needed.

For all these reasons, reference counting should be used with utmost care. You should not accept it as a default, but rather ask yourself why it is you reject what are generally better automated techniques.

Exercise

If the reference count overflows, which correctness properties are hurt and how? Examine tradeoffs.

11.4 Automated Reclamation, or Garbage Collection

Now let's briefly examine the idea of having the language's run-time system automate the process of reclaiming garbage. We'll use the abbreviation GC (for *garbage collection*) to refer to both the algorithm and the process, letting context disambiguate.

Some people call reference counting a "garbage collection" technique. I prefer to use the latter term to refer only to fully-automated techniques. But do beware this potential for confusion when browsing the Web.

11.4.1 Overview

The key idea behind all GC algorithms is to traverse memory by following references between values. Traversal begins at a *root set*, which is all the places from which a program can possibly refer to a value in the store. Typically the root set consists of every bound variable in the environment, and any global variables. In an actual working implementation, the implementor must be careful to also note ephemeral values such as references in registers. From this root set, the algorithm walks all accessible values using a variety of algorithms that are usually variations on depth-first search to identify everything that is *live* (i.e., usable through some sequence of program operations). Everything else, by definition, is garbage. Again, different algorithms address the recovery of this space in different ways.

Depth-first search is generally preferred because it works well with stack-based implementations. Of course, you might (and should) wonder where the GC's own stack is stored!

11.4.2 Truth and Provability

If you read carefully, you'll notice that I slipped an *algorithm* into the above description. This is an *implementation detail*, not part of the *specification*! Indeed, the specification of garbage collection is in terms of *truth*: we want to collect precisely all the values that are garbage, no more and no less. But we cannot obtain truth for any Turing-complete programming language, so we must settle for *provability*. And the style of algorithm described above gives us an efficient "proof" of liveness, rendering the complement garbage. There are of course variations on this scheme that enable us to collect more or less garbage, which correspond to different *strengths* of proof a value's "garbageness".

This last remark highlights a weakness of the strict specification, which says nothing about how much garbage should be collected. It is actually useful to think about the extreme cases.

Do Now!

> It is trivial to define a sound garbage collection strategy. Similarly, it is also trivial to define a complete garbage collection strategy. Do you see how?

To be sound, we simply have to make sure we don't accidentally remove anything that is live. The one way to be absolutely certain of this is to *collect no garbage at all*. Dually, the trivial complete GC collects *everything*. Obviously neither of these is useful (and the latter is certain to be highly dangerous). But this highlights that in practice, we want a GC that is not only sound but as complete as possible, while also being efficient.

11.4.3 Central Assumptions

Being able to soundly perform GC depends on two critical assumptions. The first is one about the language's implementation and the other about the language's semantics.

1. When confronted with a value, the GC needs to know what kind of value it is, and how its memory representation is laid out. For instance, when the traversal reaches a `cons` cell, it must know:

   (a) that this is a `cons` cell; and hence,

   (b) that the `first` is at, say, a four byte offset, and

   (c) that the `rest` is at, say, an eight byte offset.

   Obviously, this property must hold recursively, thus enabling a traversal algorithm to correctly map the values in memory.

2. That programs cannot *manufacture* references in two ways:

   (a) Object references cannot reside outside the implementation's pre-defined root set.

   (b) Object references can only refer to well-defined points in objects.

When the second property is violated, the GC can effectively go haywire, misinterpreting data. The first property sounds obvious: when it is violated, it seems the run-time system has clearly failed to obey the language's semantics. However, the consequences of this property are subtle, as we discuss below [REF].

11.5 Convervative Garbage Collection

We've explained that the typical root set consists of the environment, global variables, and some choice ephemerals. Where else might references reside?

In most languages, nowhere else. But some languages (I'm looking at you, C and C++) allow references to be turned into arbitrary numbers, and arbitrary numbers to be turned back into references. As a result, in principle, *any* numeric value in the program (which, because of the nature of C and C++'s types, virtually *any* value in the program) could potentially be treated as a reference.

This is problematic for two reasons. First, it means the GC can no longer limit its attention to the small root set; instead, the entire store is now potentially the root set. Second, if the GC tries to modify the object in any way—e.g., to record a "visited" bit during traversal—then it is potentially changing *non-reference* values: e.g., it might actually be changing an innocent numeric constant in the program. As a result, the particular confluence of features in languages like C and C++ conspire to make sound, efficient GC extremely difficult.

But not impossible. A stimulating line of research called *conservative* GC has managed to create reasonably effective GC systems for such languages. The principle behind conservative GC notes that, while in principle every store location might be a root, in practice many of them are not. It then proceeds through a series of increasingly clever observations to deduce what must *not* be a reference (the opposite of a traditional GC) and can hence be safely *ignored*: for instance, on a word-aligned architecture, no odd number can never be a reference. By skipping most of the store, by making some basic assumptions about program behavior (e.g., that it will not manufacture certain kinds of references), and by being careful to not modify the store—e.g., changing bits in values, or moving data around—it can actually yield a reasonably effective GC strategy.

Nevertheless, it is a bit of a dog walking on its hind legs.

Conservative GC is often popular with programming language implementations that are written in, or rely on a base of code in, C or C++. For instance, early versions of Racket relied exclusively on it. There are many good reasons for this:

1. It offers a quick bootstrapping technique, so the language implementor can focus on other, more innovative, features.
2. A language that controls all references (as Racket does) can easily create memory representations that are especially conducive to increasing the effectiveness of the GC (e.g., padding all true numbers with a one in the least-significant-bit).
3. It enables easy interoperation with useful libraries written in C and C++ (provided, of course, that they too meet the expectations of the technique).

A word on vocabulary is in order. As we have argued [REF], *all* practical GC techniques are "conservative" in that they approximate truth with reachability. The word "conservative" has, however, become a term-of-art to refer to a GC technique that operates in an *uncooperative* (and hopefully not *hostile*) run-time system.

11.6 Precise Garbage Collection

In conventional GC terminology, the opposite of "conservative" is *precise*. This, too, is a misnomer, because a GC cannot be precise, i.e., both sound and complete. Rather, precision here is a statement about the ability to identify references: when confronted with a value, a precise GC knows exactly what is and isn't a reference, and where the references are. This removes the monumental effort that a conservative GC has to put into guessing non-references (and hoping to eliminate as many potential references as possible through this process).

Within the space of precise GC, which is what most contemporary language run-time systems use, there is a wide range of implementation techniques. I refer you to Paul Wilson's survey (which, despite its relative age in this fast-moving field, remains an excellent resource), as well as the book and other materials from Richard Jones. In particular, for a quick and readable overview of a generational garbage collector, read *Simple Generational Garbage Collection and Fast Allocation*.

12 Representation Decisions

Go back and look again at our interpreter for function as values [REF]. Do you see something curiously non-uniform about it?

Do Now!

No, really, do. Do you?

Consider how we chose to represent our two different kinds of values: numbers and functions. Ignoring the superficial `numV` and `closV` wrappers, focus on the underlying data representations. We represented the interpreted language's numbers as Racket numbers, but we did not represent the interpreted language's functions (closures) as Racket functions (closures).

That's our non-uniformity. It would have been more uniform to use Racket's representations for both, or also to *not* use Racket's representation for either. So why did we make this particular choice?

We were trying to illustrate and point, and that point is what we will explore right now.

12.1 Changing Representations

For a moment, let's explore numbers. Racket's numbers make a good target for reuse because they are so powerful: we get arbitrary-sized integers (*bignums*), rationals (which benefit from the bignum representation of integers), complex numbers, and so on. Therefore, they can represent most ordinary programming language number systems. However, that doesn't mean they are what we *want*: they could be too little or too much.