Automatic Garbage Collection

Announcements:

- PS6 due Monday 12/6 at 11:59PM
- Final exam on Thursday 12/16
    - PS6 tournament and review session that week

Garbage

- In OCaml programs (and in most other programming languages), it is possible to create **garbage**: allocated space that is no longer usable by the program.
- For example, consider this code:

```
let x =[[1;2;3];[4]] in
let y = [2]::List.tl x in
   y
```

- Here the variable `x` is bound to a list of two elements, each of which is itself a list.
- Then the variable `y` is bound to a list that drops the first element of `x` and adds a different first element and the function returns this new list.
- Since `x` is never used again that first list is inaccessible, or garbage.

Reachability

- Any boxed value created by OCaml can become garbage.
- This includes tuples, records, strings, arrays, lists and function closures as well as most user-defined data types.
- Most garbage collectors are based on the idea of reclaiming whole blocks that are no longer **reachable** from a set of **roots**,
    - which are pointers into the heap that are assumed to always be accessible.
- The roots of a given computation consist of pointers that appear in the environment, plus the pointer to the currently computed result.
- A block of memory is reachable from the roots if there is a direct pointer to that block among the roots,
    - or if there is a pointer to that block in another block that is reachable from the roots.
- Looking a memory more abstractly, we see that the memory heap is simply a directed graph in which the nodes are blocks of memory and the edges are the pointers between these blocks.
- So reachability can be computed as a graph traversal.

Circular data structures

- A particularly challenging type of data structure
  - Extremely popular in interview questions
  - Likely to appear on CS3110 final exam

- Motivation: suppose you need a buffer, for instance to hold the last 100 stock quotes on the NYSE
  - Some other thread is reading them and discarding them
  - Similar issues for, e.g., audio/video buffering

- Natural solution is a queue, which is ideal for unbounded input
- But our input size is bounded, and we can exploit this
- Make an nlist with 100 elements, and have the last one's tail point to the first one
- Tiny example:

```
type nlist = Nnil | Ncons of (int * nlist ref)

let n1 = Ncons(1, ref Nnil)

let Ncons(a,b) = n1 in b := n1
```

- We need to keep track of where we are adding data and where we are removing data (i.e., the tail and head of the queue)
- Simple questions: how many elements does this circular list have?
  - Not trivial, need to use == instead of =
- Harder questions: efficiently detect or reverse circular nlists

Explicit vs. automatic garbage collection

- There are two basic strategies for dealing with garbage:
    - explicit garbage collection by the programmer,
    - and automatic garbage collection built into the language run-time system.
- Explicit garbage collection is provided by languages like C and C++.
- There is a way to explicitly deallocate (or "free") allocated memory when it is expected that that memory is about to become garbage.
- Languages like Java and OCaml provide automatic garbage collection :
- the system automatically identifies blocks of memory that can never be used again by the program,
    - and reclaims their space for use by later allocations.

- Automatic garbage collection offers the advantage that the programmer does not have to worry about when to deallocate a given block of memory.
- In languages like C the need to explicitly manage memory complicates any code that allocates data on the heap, and is a significant burden on the programmer.
- Worse, if the programmer fails to deallocate properly, bugs are introduced into the program that are hard to find:

    - If the programmer neglects to deallocate some garbage, it creates a memory leaks in which some allocated memory can never again be reused.
    - This is a program for long-running programs which will tends to grow in size until they consume all of memory.

    - If the programmer is too aggressive and deallocates a block of memory that is still in use, this creates a **dangling pointer** that may be followed later even though it now points to unallocated memory or to a new allocated value that may be of a different type.

    - If a block of memory is deallocated *twice*, this typically corrupts the memory heap data structure even if the block was initially garbage.
    - Corruption of the memory heap is likely to cause unpredictable effects later during execution and be difficult to debug.

- In practice, programmers manage explicit allocation and deallocation by keeping track of what piece of code "owns" each pointer in the system.
- That piece of code is responsible for deallocating the pointer later.
- The tracking of pointer ownership shows up in the specifications of code that manipulates pointers, complicating specification, and use, and implementation of the abstraction.

- Automatic garbage collection helps modular programming, because two modules can share a value without having to agree on which module is responsible for deallocating it.
- The details of how boxed values will be managed does not pollute the interfaces in the system.

Requirements for automatic garbage collection

- Many programs written in OCaml (and Java) generate garbage at a high rate, so it is important to have an effective way to collect the garbage. The following properties are desirable in a garbage collector:
    - It should identify most garbage
    - Anything it identifies as garbage must be garbage **[SOUNDNESS]**
    - It should impose a low added time overhead
    - During garbage collection the program may be paused; these pauses should be short
- Fortunately modern garbage collectors provide all of these important properties. We will not have time for a complete survey of modern garbage collection techniques, but we can look at some simple garbage collectors.

Identifying pointers

- To compute reachability accurately, the garbage collector needs to be able to identify pointers; that is, the edges in the graph.
- Since a word of memory cells is just a sequence of bits, how can the garbage collector tell apart a pointer from an integer?

- One simple strategy is to reserve a bit in every word to indicate whether the value in that word is a pointer or not.
- This **tag bit** uses up about 3% of memory, which may be acceptable. It also limits the range of integers (and pointers) that can be used.
- On a 32-bit machines, using a single tag bit means that integers can go up to about 1 billion, and that the machine can address about 2GB instead of the 4GB that would otherwise be possible.
- Adding tag bits also introduces a small run-time cost that is incurred during arithmetic or when dereferencing a pointer.

- A different solution is to have the compiler record information that the garbage collector can query at run time to find out the types of the various locations on the stack.
- Given the types of stack locations, the successive pointers can be followed from these roots and the types used at even step to determine where the pointers are.
- This approach avoids the need for tag bits but is substantially more complicated because the garbage collector and the compiler become more tightly coupled.

- Finally, it is possible to build a garbage collector that works even if you can't tell apart pointers and integers.
- The idea is that if the collector encounters something that looks like it might be a pointer, it treats it as if it is one, and the memory block it points to is treated as reachable.
- Memory is considered unreachable only if there is nothing that looks like it might be a pointer to it.

- This kind of collector is called a **conservative** collector because it may fail to collect some garbage, but it won't deallocate anything but garbage.
- In practice it works pretty well because most integers are small and most pointers look like large integers.
- So there are relatively few cases in which the collector is not sure whether a block of memory is garbage.

Mark and sweep collection

- Mark-and-sweep proceeds in two phases:
    - a mark phase in which all reachable memory is marked as reachable,
    - and a sweep phase in which all memory that has not been marked is deallocated.
- This algorithm requires that every block of memory have a bit reserved in it to indicate whether it has been marked.

- Marking for reachability is essentially a graph traversal; it can be implemented as either a depth-first or a breadth-first traversal, though depth-first traversal is likely to be faster.
- One problem with a straightforward implementation of marking is that graph traversal takes O(n) space where n is the number of nodes.
- However, this is not as bad as the graph traversal we considered earlier, one needs only a single bit per node in the graph if we modify the nodes to explicitly mark them as having been visited in the search.
- Nonetheless, if garbage collection is being performed because the system is low on memory, there may not be enough added space to do the marking traversal itself.

- A simple solution is to always make sure there is enough space to do the traversal.
- A cleverer solution is based on the observation that there is O(n) space available already in the objects being traversed.
- It is possible to record the extra state needed during a depth-first traversal on top of the pointers being traversed.
- This trick is known as **pointer reversal**. It works because when returning from a recursive call to the marking routine, the code knows what object it came from.

- Therefore, the predecessor object that pointed to it does not need the word of storage that it was using to store the pointer; it can be restored on return.
- That word of storage is used during the recursive call to store the pointer to the predecessor's predecessor, and so on.
- In the sweep phase all unmarked blocks are deallocated. This phase requires the ability to find all the allocated blocks in the memory heap, which is possible with a little more bookkeeping information per each block.

Triggering garbage collection

- When should the garbage collector be invoked?
- An obvious choice is to do it whenever the process runs out of memory.
- However, this may create an excessively long pause for garbage collection.
- Also, it is likely that memory is almost completely full of garbage when garbage collection is invoked.
- This will reduce overall performance and may also be unfair to other processes that happen to be running on the same computer.
- Typically, garbage collectors are invoked periodically, perhaps after a fixed number of allocation requests are made, or a number of allocation requests that is proportional to the amount of non-garbage (**live**) data after the last GC was performed.

Reducing GC pauses

- One problem with mark-and-sweep is that it can take a long time -- it has to scan through the entire memory heap.
- While it is going on, the program is usually stopped.
- Thus, garbage collection can cause long pauses in the computation.
- This can be awkward if, for example, one is relying on the program to, say, help pilot an airplane.
- To address this problem there are **incremental** garbage collection algorithms that permit the program to keep computing on the heap in parallel with garbage collection, and **generational** collectors that only compute whether memory blocks are garbage for a small part of the heap.

Compacting (copying) garbage collection

Collecting garbage is nice, but the space that it creates may be scattered among many small blocks of memory. This external fragmentation may prevent the space from being used effectively. A compacting (or copying) collector is one that tries to move the blocks of allocated memory together, compacting them so that there is no unused space between them. Compacting collectors tend to cause caches to become more effective, improving run-time performance after collection.

Compacting collectors are difficult to implement because they change the locations of the objects in the heap. This means that all pointers to moved objects must also be updated. Finding all these pointers can be expensive and requires added storage or time.

Some compacting collectors work by using an **object table** containing pointers to all allocated objects. Objects themselves only contain pointers into (or indices of) the object table. This solution makes it possible to move all allocated objects around because there is only one pointer to each object. However, it doubles the cost of following a pointer.

Reference counting

A final technique for automatic garbage collection that is occasionally used is **reference counting**. The idea is to keep track for each block of memory how many pointers there are incoming to that block. When the count goes to zero, the block must be unreachable and can be deallocated.

There are a few problems with this conceptually simple solution:

- It imposes a lot of run-time overhead, because each time a pointer is updated, the reference counts of two blocks of memory must be updated (one incremented, one decremented). This cost can be reduced by doing compile-time analysis to determine which increments and decrements are really needed.
- It can take a long time, because deallocating one object can cause a cascade of other objects to be deallocated at the same time. The solution to this problem is to put objects to be deallocated onto a queue. When an allocation for *n* bytes is performed, objects taking up space totaling at least *n* bytes are dequeued and deallocated, possibly causing more objects to lose all their references and be enqueued.
- It cannot collect garbage that lies in a **cycle** in the heap graph, because the reference counts will never go down to zero. Cyclical data structures are common, for instance with many representations of directed graphs.

Generational garbage collection

Generational garbage collection separates the memory heap into two or more **generations** that are collected separately. In the basic scheme, there are tenured and new (untenured) generations. Garbage collection is mostly run on the new generation (minor collections), with less frequent scans of older generations (major collections). The reason this works well is that most allocated objects die young; in many programs, the longer an object has lasted, the longer it is likely to continue to last. Minor collections are much faster because they run on a smaller heap. The garbage collector doesn't waste time trying to collect long-lived objects.

After an allocated object survives some number of minor garbage collection cycles, it is promoted to the tenured generation so that minor collections stop spending time trying to collect it.

Generational collectors introduce one new overhead. Suppose a program mutates a tenured object to point to an untenured object. Then the untenured object is reachable from the tenured set and should not be collected. The pointers from the tenured to the new generation are called the **remembered set**, and the garbage collector must treat these pointers as roots. The language run-time system needs to detect the creation of such pointers. Such pointers can only be created by imperative update; that is the only way to make an old object point to a newer one. Therefore, imperative pointer updates are often more expensive than you might expect. Of course, a functional language like OCaml discourages these updates, which means that they are usually not a performance issue.

Copying collectors

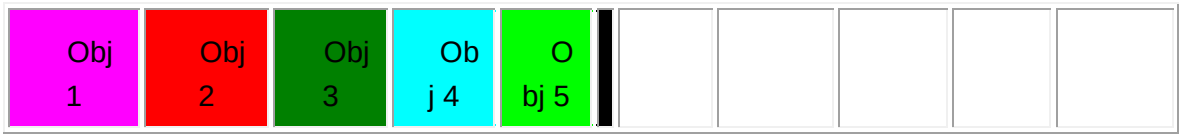
The goal of a garbage collector is to automatically discover and reclaim fragments of memory that will no longer be used by the computation. It turns out that during evaluation of a

high-level program, we allocate lots of little objects that are only in use for short periods of time and can be effectively recycled.

Most garbage collectors are based on the idea of reclaiming whole objects that are no longer*reachable* from a *root set*. In the case of our interpreter, we only access memory objects through the stack and through pointers. So any object that isn't reachable from the stack, following the pointers contained within other objects, can be safely reclaimed by a garbage collector.

At an abstract level, all a *copying collector* does is start from a set of roots (in our case, the operand stack), and traverse all of the reachable memory-allocated objects, copying them from one half of memory into the other half. The area of memory that we copy from is called **old space** (or **from-space**) and the area of memory that we copy to is called **new space** (or **to-space**). When we copy the reachable data, we **compact** it so that it is in a contiguous chunk. So, in effect, we squeeze out the holes in memory that the garbage data occupied. After the copy and compaction, we end up with a compacted copy of the data in new space data and a (hopefully) large, contiguous area of memory in new space in which we can quickly and easily allocate new objects. The next time we do garbage collection, the roles of old space and new space will be reversed.

For example, suppose memory looks something like this, where the colored boxes represent different objects, and the thin black box in the middle represents the half-way point in memory.



At this point, we've filled up half of memory and so we initiate a collection. Old space is on the left and new space on the right. Suppose further that only the red and light-blue boxes (objects 2 and 4) are reachable from the stack. After copying and compacting, we would have a picture like this:

Notice that we copied the live data (the red and light-blue objects) into new space, but left the unreachable data in the first half. Now we can "throw away" the first half of memory (this doesn't really require any work):

After copying the data into new space, we restart the computation where it left off. The computation continues allocating objects, but this time allocates them in the other half of

memory (i.e., new space). The fact that we compacted the data makes it easy for the interpreter to allocate objects, because it has a large, contiguous hunk of free memory. So, for instance, we might allocate a few more objects:

When the new space fills up and we are ready to do another collection, we flip our notions of new and old. Now old space is on the right and new space on the left. Suppose now that the light-blue (Obj 4), yellow (Obj 6), and grey (Obj 8) boxes are the reachable live objects. We copy them into the other half of memory and compact them, throwing away the old data:

What happens if we do a copy but there's no extra space left over? Typically, the garbage collector will ask the operating system for more memory. If the OS says that there's no more available (virtual) memory, then the collector throws up its hands and terminates the whole program.

Implementation Details (for the interested)

It is surprisingly simple to build a copying garbage collector.

First, we need one more register `scanptr`, which will be used as an index into memory. The`scanptr` initially contains the base address of the new space (*i.e.*, the address of the first word where we will copy objects.) We also set the `allocptr` to the base address of the new space. We will use the `allocptr` to remember where to allocate objects as we copy them from old space to new space. The purpose of the `scanptr` will be made clear below.

Second, starting from an array of roots (in our case, the values pushed on the operand stack), we examine each root to see whether or not it's a pointer. If so, we copy the object that the pointer references from old space to new space. We can figure out how big the object is, because it always starts with a `Tag_v` value specifying its length (but see below.) As we copy the object, we place it where the `allocptr` currently is, and then increment the `allocptr` by the appropriate amount so that it points to the next available spot in new space.

After we copy the object, we need to update the array of roots so that it points to the new copy of the object. We also need to leave a forwarding pointer in the object's previous location in old space indicating that the object has already been moved and where it was moved to, in case we run into this particular object again while traversing the graph of reachable objects in old space. (This prevents us from going into an infinite loop if there are cycles in the graph, and there typically will be cycles if we use recursive functions.) We can

do this by overwriting the **`Tag_v`**value on the old copy of the object with a forwarding pointer, represented in our abstract machine by **`Forward_v(i)`** values, where **`i`** is the address in new space of the new copy of the object.

Before trying to copy any object, we should check first to see if the object has already been forwarded. If so, then we should update the root with the address of the copied object that we find in the forwarding pointer.

So, we need a procedure **`forward`** which, when given an address *a,* (1) checks to see if the object at *a* has been forwarded or not -- if so, **`forward`** immediately returns the address of the forwarded object, (2) otherwise, copies the object from old space to new space, incrementing**`allocptr`** appropriately, (3) overwrites the old objects **`Tag_v`** with a forwarding pointer to the new copy, (4) returns the address of the new copy of the object.

Processing the roots then becomes a simple loop which simply checks to see of a root value is a pointer, and if so, calls forward, and updates the root array with the new address of the object.

After processing the roots, you've managed to copy all of the data that are immediately reachable from the roots. However, we must also process all of the data that are reachable from these objects (and then all the data reachable from those objects, and so on.) This is the purpose of the **`scanptr`** register.

After forwarding the root objects, we must then examine each of their components to see if there are any pointers. Any pointers in those objects must also be forwarded from old space to new space. The process of examining a copied object for pointers and forwarding those objects is called *scanning* the object.

The **`scanptr`** register is used to keep track of which objects have been forwarded but not yet scanned. In particular, the **`scanptr`** starts off as the same address as the **`allocptr`**. After forwarding all of the roots, we start scanning the object pointed to by the **`scanptr`**. This will be the first object that we copied during the root processing. After scanning this object, we increment the scan pointer so that it points to the next object to be scanned. The scan pointer thus moves only from left to right.

During scanning, we need to look at each value in the object. If the value is a pointer, it should be forwarded. This may cause the **`allocptr`** to move. But because the newly copied object comes after the object being scanned, we will eventually scan it and any objects that it references will also be copied into new space.

The whole process stops when the **`scanptr`** catches up to the **`allocptr`**, for then all reachable objects have been successfully forwarded from old space to new space. At this point, we need to reset the **`limitptr`** and the **`startptr`** appropriately. We also need to check that there is enough free space for the computation to make progress. If not, then

an `OutOfMemory`exception should be raised. (Otherwise, the interpreter will go into an infinite loop trying to garbage collect forever.)

Note that we are effectively using a queue to keep track of those objects in the graph that have been forwarded but not yet scanned. We insert items at the end of the queue (where `allocptr`points) and remove objects to be scanned from the front of the queue (where `scanptr` points). It's possible to use a different data structure such as a stack to keep track of those objects that have been copied but not yet scanned, but doing so usually requires an extra hunk of memory.

Note that by using a queue in the graph traversal, the copying collection effectively does a**breadth-first** traversal of the data. If we used a stack instead of a queue, the traversal would be**depth-first**.

It should also be noted that what we have described is a fairly simple take on garbage collection. There are many different algorithms, such as Mark and Sweep, Generational, Incremental, Mostly-Copying, etc. Often, a good implementation will combine many of these techniques to achieve good performance. You can learn about these techniques in a number of places---perhaps the best place to start is the **Online GC FAQ.**

Real-world garbage collectors

OCaml uses a hybrid generational garbage collector, where small objects are created and managed on one heap (minor) and large objects on another heap (major). The minor heap is collected frequently and the major less frequently. Objects that exist for sufficiently long in the minor heap are moved to the major one.

The Java5 garbage collector is also a generational collector. In the two youngest generations, a copying collector is used. However, a *mark-compact* collector manages the third and oldest generation. This collector marks the live objects in the heap, then slides them toward the beginning of the heap, overwriting any garbage in the process. Java 5 also makes two other collectors available: a concurrent copying collector for young generations, and a concurrent mark-sweep collector for the old generation.