Lecture Notes on Garbage Collection

15-411: Compiler Design
Frank Pfenning

Lecture 22
April 4, 2024

These brief notes only contain a short overview, a few pointers to the literature with detailed descriptions, and a few remarks particularly relevant to C0.

1 Introduction

So far, in C0 we have had only primitives for *allocation* of memory on the heap. Memory was never freed. In C, the `free` function accomplishes this, but it is very error-prone. Memory may not be freed that is no longer needed (a *leak*), or, worse, memory may be free that will be referenced later in the computation. In type-safe languages this can avoided by using *garbage collection* that automically reclaims storage that can no longer be referenced. Since it is undecidable if memory might still be referenced, a garbage collector uses a conservative approximation, where different techniques may approximate in different ways.

There are three basic garbage collection techniques.

Reference Counting. Each heap object maintains an additional field containing the number of references to the object. The compiler must generate code that maintains this reference count correctly. When the count reaches 0, the object is deallocated, possibly triggering the reduction of other reference counts. Reference counts are hard to maintain, especially in the presence of optimizations. The other problem is that reference counting does not work well for circular data structures because reference counts in a cycle can remain positive even though the structure is unreachable. Nevertheless, reference counting appears to remain popular for scripting languages like Perl, PHP, or Python. Another use of reference counting is in part of an operating system where we know that no circularities can arise.

Mark-and-Sweep. A mark-and-sweep collector traverses the stack to find pointers into the heap and follows each of them, marking all reachable objects. It then

sweeps through memory, collecting unmarked objects into a free list while leaving marked objects in place. It is usually invoked if there is not enough space for a requested allocation. Because objects are never moved once allocated, a mark-and-sweep collector runs the risk of fragmented memory which can translate to poor performance. Another difficulty with a mark-and-sweep collector is that the cost of a collection is proportional to all available memory (which will be touched in the sweep phase).

Copying Collection. A copying collector also traverses the heap, starting from the so-called *root pointers* on the stack. Instead of marking objects it moves reachable objects from the heap to a new area called the *to-space*. When all reachable objects have been moved, the old heap (the *from-space*) and the to-space switch roles. The copying phase will compact memory, leading to good locality of reference. Moreover, the cost is only proportional to the *reachable* memory rather than all allocated memory. On the other hand, a copying collector typically needs to run with twice as much memory than a mark-and-sweep collector.

Mark-and-sweep and copying collectors are called *tracing* collectors, since they determine the reachable (or *live*) objects on the heap by following pointers. They tend to suffer from long pauses when a garbage collection is performed. Many variations and refinements have been proposed to overcome some of the difficulties and drawbacks in various forms of garbage collectors. A somewhat dated, but still excellent survey on garbage collection by Wilson[1] [Wil92].

For this course and the C0 language we recommend implementing a simple copying collector. Experience shows that it is less error-prone and easier to implement than a mark-and-sweep or copying collector. Since there is extensive and accessible literature on garbage collection, in the remainder of this note we focus on the compiler support that is necessary for a tracing collector. The issues for mark-and-sweep and copying collectors are very similar.

We explicit do not discuss many optimizations and refinements of the basic schemes. Some of these are common sense, others can be found in the literature.

2 Allocation

In a mark-and-sweep collector, allocation is handled via a free list, usually doubly linked. This is analogous to the data structure maintained by implementations of malloc/free in C. When allocation is called we traverse the free list until we find a block that is big enough for the requested payload plus header information. We return a pointer to this object (usually with the header to the left of the pointer) which should be aligned at 0 modulo 8. If a sufficiently large portion of the block

[1] Revised version at http://www.cs.cmu.edu/~fp/courses/15411-f13/misc/wilson94-gc.pdf

is unused, it is returned to the free list for further allocations. We run out of space if we cannot find any block that is big enough.

In a copying collector there is no free list. Instead we have a currently used half-space and a next pointer to the end of the currently used portion of the half-space. We return a pointer to its beginning (perhaps after adding a fixed offset to allow for a header) and advance the next pointer. Allocation in a copying collector is typically significantly faster than in a mark-and-sweep collector.

3 Finding Root Pointers

Assume that either `alloc` or `alloc_array` is called and we have run out of space. We now need to find the *root pointers* on the stack that point to the heap. This task is simplified in C0 since we have no pointers to the stack, unless the compiler optimizes to allocate some data on the stack. The compiler lays out each stack frame, so it knows where to find pointers and what their types are. Which pointers exactly are still live (and even where they are on the stack) may change during the computation of the function. We therefore best associate this information with each *return address*.

The information we need at a snapshot of the stack frame is the place where pointers are. This may be kept in a *pointer map* for the stack frame. A reference to the pointer map may be kept in a seperate data structure, or in the stack frame itself (for example, at the bottom or top of the stack frame).

Since have the traverse the stack it is convenient to keep base pointers for each frame which are pushed onto the stack just as in the x86 calling convention. While optional for the x86-64, it is quite useful for the garbage collector. So, once we have processed the pointers in the current frame, we find the return address to get the pointer map for the previous frame until we get to the first frame on the stack.

There are some subtleties regarding registers. Since the last call will always be to `alloc` or `alloc_array` we don't have to worry about registers except callee-save registers. The called function will not be able to tell if any of the callee-save registers contains a root pointer. A simple strategy to handle such callee-save registers is for the caller (who knows what they are) to simply push callee-save registers containing live root pointers onto the stack and add these locations to the pointer map. They do not need to be restored, due to the callee-save protocol. Caller-save and argument registers will already be on the stack (and in the pointer map) if they are live.

4 Derived Pointers

Some computations and optimizations will compute addresses of data in the middle of heap objects. It is important that we don't follow such pointers since, for

example, the header of an object will not be at a fixed offset from such a derived pointer. Instead, the compiler should arrange to keep mark the corresponding base pointer as *live* and keep it somewhere were it is recognized as a root pointer. A typical example of this is the computation of the address of an array element, which will be a derived pointer.

5 Traversing the Heap

When we traverse the heap, whether just marking or copying reachable objects, we need to identify pointers in the objects we reach. The traditional way to accomplish this is to keep a reference to a pointer map in the object header. Just as for a stack frame, every pointer in the heap object has an entry with its offset in the pointer map, with a special indicator for arrays. In addition the header should have a bit that can marked when visited (for mark-and-sweep) or marked when copied (for a copying collector). In the case of a copying collector, we need to make sure the object is big enough to hold the *forwarding pointer* when it is moved to the *to-space*. In the case of a mark-and-sweep collector the object has to be big enough to hold the next and prev pointers of the doubly-linked free list.

6 Tagless Garbage Collection

In a safe, statically typed language such as C0, pointer maps are not strictly necessary for heap objects as long as we can determine the types of the root pointers. After that, the type of every heap object we reach, and the types of the components, are determined entirely from the types and the computed offsets for structs.

As suggested by Goldberg[2] [Gol91] an efficient and convenient way to achieve this is for the compiler to generate a structure traversal function for each type that may be allocated on the heap. For each root pointer we then just need to know which garbage collection traversal function to call. This is something the compiler can know (since types are known at compile-time) and store at an appropriate place, either in the stack frame, the text segment, or allocate during an initialization phase in a global variable.

References

[Gol91] Benjamin Goldberg. Tag-free garbage collection fro strongly typed programming languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI'91, pages 165–176. ACM, June 1991.

[2] and a student during lecture

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM'92, pages 1–42, London, UK, 1992. Springer-Verlag.