Comp 411
Principles of Programming Languages
Lecture 27
Storage Management

Corky Cartwright

April 9, 2021

Heap Management

In all of our interpreter designs (including those written in machine code or C), we have presumed the existence of a heap supporting dynamic allocation (`new` operations in C++ or Java).

For efficient space utilization, the heap must reclaim storage that is no longer in use by the program. In C and C++, the programmer is responsible for explicitly reclaiming storage (the `free` and `delete` operations in C and C++, respectively). In a well-written C/C++ program, all dynamically allocated storage is freed just before it becomes inaccessible (no object becomes inaccessible prior to being freed; once freed, an object is never referenced again).

In practice, manual storage management is clumsy (interfaces become much more complex) and error-prone.

Automatic Heap Management

- Two fundamental approaches to automatic heap management:
- *Reference counting*: every object includes a field that counts the number of objects (every data structure containing a heap pointer including an ordinary variable is called an object) pointing to it. Some schemes defer updating reference counts, but an object with a deferred count cannot be freed.
- *Garbage collection*: periodically (usually when no free space is left) the heap management system determines which objects in the heap have become inaccessible.
- The term *garbage collection* is not used consistently in the literature. In some cases, it means any approach to automatic heap management. In others it refers to schemes that rely on pointer tracing rather than reference counting. I will use the latter convention.

Reference Counting

- Every object includes an additional field that counts the number of objects pointing to it; this field must be large enough so that it cannot overflow (*e.g.*, machine word [address] size).
- When an object is created, its reference count is set to 1 (and a pointer to it must be created within some other object, perhaps a variable).
- When a pointer field in an object is mutated, the reference count for the old object is decremented and the reference count for the new object is incremented. This combined operation must be atomic with respect to the checking for 0 reference counts.
- When a reference count for an object becomes 0, the object is *freed* (returned to free storage).
- There are many possible optimizations; reference counting for local storage is expensive and can be greatly reduced by static analysis and the use (by the programmer) of *weak pointers* that are ignored in reference counting. A weak pointer is a redundant pointer to an object; the storage management system recognizes when a weak pointer is no longer valid (points to an object that has been deallocated) and sets its value to **`null`** (or *None* in frameworks with “option” types). Some compilers implementing languages that mandate the use of reference counting in storage management (like Objective C and Swift) aggressively optimize generated code.