15-411 Compiler Design, Fall 2021
Lab 6: Garbage Collection

Seth and co.

Refer to main Lab 6 handout for deadlines

1 Introduction

This writeup describes the option of implementing garbage collection; it is intended as an extension to the instructions given in the main Lab 6 handout.

The language L4 does not change for this lab and remains the same as in Labs 4 and 5.

2 Components

2.1 Garbage Collecting Compiler and Runtime

Your compilers should treat the language L4 as in Labs 4 and 5. It is not necessary to support the `--unsafe` flag or any optimization flags for this assignment. You have complete freedom with regard to which kind of garbage collector to implement. A garbage collector will consist of the compiler proper and the runtime system. The interface from the compiled code to the runtime system should be part of your design. Reasonable choices are a mark-and-sweep or a copying collector, but even a conservative collector is acceptable. Incremental or generational collectors are significantly harder and should only be attempted if you already have a basic collector working.

Grading criteria includes:

1. Functional correctness is paramount. You should not mutate the heap in such a way as to result in the incorrect execution of programs. Your compiler should continue to function correctly, despite any changes to the binary interface you use to interface with the garbage collector.

2. Developing a convincing framework for *understanding* and *quantifying* the performance of your garbage collector is also a priority.

3. The absence of memory leaks comes second. A garbage collector that takes a whole lot of processor time is not of much use if it cannot effectively collect parts of the heap that are no longer referenced. However, conservative collectors may not be able to reclaim all memory.

4. Performance is a distant third – actual performance has a very minor effect on your grade. Optimizations to garbage collectors require a significant amount of time. Therefore, we recommend that you avoid premature optimizations.

2.2 Tests and Measurement Tools

At minimum, your testing *must* be sufficient to provide strong evidence of two things:

1. Your garbage collector does not corrupt the heap, and does not leak memory.
2. Your garbage collector allows programs to run that would otherwise have failed due to lack of resources.

To this end, feel free to search through all of the test cases that we have accumulated through this semester for programs that are both realistic and usefully contrived to assemble a test suite. You will need to write contrived test cases designed specifically to ensure that your garbage collector goes through several collection cycles without corrupting the heap or leaking any memory. You will also need to run your compiled programs in an environment where memory is artificially constrained. You will be graded on how well you test your garbage collector.

2.3 Something Extra

Beyond the basic implementation and perfunctory analysis described above, an excellent final project will include a little something extra. Exactly what your "something extra" is is up to you, but it should represent a different direction, rather than (just) being an incremental or generational version of the collector you implemented.

One category of "something extra" ideas involves going beyond the perfunctory analysis described above and doing a significant quantitative analysis of your garbage collector's performance. These are couple of (non-exhaustive) suggestions:

- Compare the performance of multiple garbage collection strategies. Note that this requires implementing multiple garbage collection strategies, which is rather ambitious!
- Compare the performance of your compiler to the Boehm-Demers-Weiser conservative collector used by the reference compiler, which can be used as a drop-in replacement for `malloc()`. You can do this either by compiling with the reference compiler and runtime or by obtaining the Boehm-Demers-Weiser collector (see http://www.hboehm.info/gc/).
- Extend L4 with manual memory management using a built-in `free()` function that accepts any pointer or array type. Compare the performance of the manual and garbage-collected memory management.
- Analyze the way in which performance is influenced by varying the total amount of memory available to your program. (You can measure performance both in terms of total running time and in terms of total number of collections.)

Another way to do something extra is to explore a language feature that is enabled by garbage collection:

- Implement *weak pointers*, references that do not keep the allocations they point to from being reclaimed. In order to preserve memory safety, it must always be possible to determine if the target of a weak pointer has been reclaimed. Demonstrate an interesting program using weak pointers. (See section 7.1 of the Wilson review.)

- Implement *finalizers*, functions that are associated with a specific pointer or a specific type of pointer and that run when the pointer is freed by garbage collection. Demonstrate an interesting program using finalizers. (See section 7.2 of the Wilson review.)
- Implement a leak detector: use your garbage collection infrastructure to explore a safe implementation of `free()` that always safely detects double-frees (signaling a memory error) and reports the number of un-freed allocations when the program exits.

3 Requirements, Deliverables, and Deadlines

The requirements of a working compiler with documentation, testing suite, runtime system, and term paper are not different from the Create-Your-Own-Adventure option for Lab 6. Please see the main Lab 6 handout on the course website for instructions.

4 Notes and Hints

- Limit optimizations. Garbage collection is easier if fewer optimizations are applied to the code, especially where memory references are concerned. In order to concentrate on the garbage collector, it is probably a good idea to stay away from optimizations altogether.
- Copying vs. mark-and-sweep collector. Experience in previous years indicates that a copying collector is easier to implement for our language than a mark-and-sweep collector because the data structures are simpler.