15-411 Compiler Design, Fall 2015
Lab 6 - Garbage Collection

Rob Simmons, Will Crichton, Grant Della Silva, Matt Bryant, Anshu Bansal

Due 11:00pm, Thursday, December 10, 2015
Papers due 11:00pm, Tuesday, December 15, 2015

1 Introduction

The main goal of the lab is to explore advanced aspects of compilation. This writeup describes the option of implementing garbage collection; other writeups detail other options. The language L4 does not change for this lab and remains the same as in Labs 4 and 5.

2 Requirements

You are required to hand in three separate items:

1. The working compiler and runtime system that implement optimizing transformations,
2. a testing framework, and
3. a term paper describing and critically evaluating your project.

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 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* 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.

2.4 Term Paper

Your paper should follow this outline.

1. Introduction. This should provide an overview of your implementation and briefly summarize the results you obtained.
2. Compilation. Describe the data structures, code, and information generated by the compiler in order to support the garbage collector. If applicable, describe the syntax and semantics of new language features.
3. Runtime System. Describe the runtime system of the garbage collector, giving details of the algorithms and also its implementation (most likely in C).
4. Testing Methodology. Describe the criteria based on which you selected and designed your tests, and explain how you use them to verify the functionality of your garbage collector.
5. Analysis. Critically evaluate your collector and sketch future improvements one might make to its basic design.

The term paper will be graded. There is no hard limit on the number of pages, but we expect that you will have approximately 5-10 pages of reasonably concise and interesting analysis to present.

3 Deadlines and Deliverables

All your code should be placed in subdirectories of the `lab6gc` directory as before. The autograder will run regression tests against your own tests and the same tests it used in Lab 5, but these will *not* directly contribute to your grade. (In particular, we don't care if there are lots of timeouts.) We will grade you based on the code and `README` file(s) you have checked in at the deadline.

Compiler Files (due 11:00pm on Thu Dec 10)

All files should be collected in a directory `compiler/` which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a roadmap to your code. This will be a helpful guide for the grader.

Issuing the shell command

```
% make
```

should generate the appropriate files so that

```
% bin/c0c --exe <args>
```

will run your compiler in safe mode with support for garbage collection. It is not necessary to continue supporting any compiler flags besides `-t`.

After running `make`, issuing the shell command

```
% make test
```

should run all your own tests and print out informative output. The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

Runtime environment (due 11:00pm on Thu Dec 10)

You will need to extend the `run411.c` runtime file and/or write new runtime files to implement garbage collection. All necessary runtime files should be included in the `lab6gc/compiler/` directory (or a subdirectory) for this assignment.

Because we don't know exactly how your runtime works, your compiler should have an additional flag `--exe`. If compiler is given a well-formed input file `foo.l1` or `foo.l2` as a command-line argument and is also given the `--exe` argument, it should generate a target file called `foo.l1.s` or `foo.l2.s` (respectively) in the same directory as the source file, and should *also* compile the garbage collector and runtime and link your generated assembly to create an executable `foo.l1.exe` or `foo.l2.exe` (respectively).

Your `README` should explain how your compiler implements the `--exe` option, and should also explain how to link other runtimes and how to use any other command-line options used by the compiler or by the generated programs. For example, you may want a command-line argument accepted by either the compiler or the generated executable that determines how much heap space the executable is allowed to use.

Tests and Measurement Tools (due 11:00pm on Thu Dec 10)

All tests you develop should be submitted in a directory called `lab6gc/tests/`. If they are to be used in a different way than a vanilla L4 test, you should include a `README` file explaining exactly how to use your tests, and `make test` should run your tests.

If you modify the autograder driver as part of your testing, or develop any other shell scripts that facilitate testing, you should include these in the `lab6gc/compiler/` directory (or a subdirectory). If there are any special instructions we need to follow in order to be able to run the driver on your compiler and test it, specify these instructions in your `README` file.

3.1 Term Paper (due 11:00pm on Tue Dec 15)

Submit your term paper in PDF form via Autolab before the stated deadline. Early submissions are much appreciated since it lessens the grading load of the course staff near the end of the semester. **You may not use any late days on the term paper describing your implementation of Lab 6!**

4 Notes and Hints

- Limit optimizations. Garbage collection is easier if fewer optimization 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.
- Apply regression testing. It is very easy to get caught up in the new functionality.
- 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.