Assignment 5: Writing a Garbage Collector for Feeny

Patrick S. Li

October 2, 2015

1 Introduction

In this assignment, you will implement a compacting precise garbage collector for automatically reclaming storage during the execution of Feeny programs. You will be implementing a *stop-and-copy* or *scavenging* garbage collector on top of your bytecode interpreter. This is the last step required before you have a *correct* (though slow) Feeny implementation.

2 Harness Infrastructure

You will be expected to execute Feeny programs from a given AST, and will be using the harness from the bytecode compiler assignment.

The function

```
Program* compile (ScopeStmt* stmt)
```

in the file `src/compiler.c` will be the entry point of your bytecode compiler, and will be called with the AST datastructure.

As in the last assignment

```
void interpret_bc (Program* p)
```

in the file `src/vm.c` will be the entry point of your bytecode interpreter and will be called with the result of your bytecode compiler.

2.1 Test Harness

The provided bash script `run_tests` will parse the test programs in the `test` directory into ASTs and run your bytecode compiler and interpreter on the result. To run it, type:

```
./run_tests compiler.c vm.c
```

at the terminal. For each test program, e.g. `cplx.feeny`, the output of your bytecode interpreter will be stored in `output/cplx.out`. The default implementations of `compile` and `interpret_bc` do nothing except print out the AST and bytecode IR respectively.

Please ensure that your implementation can be driven correctly by `run_tests`. Your assignment will be graded using this script.

2.2 Compiling and Running Manually

To compile and run your implementation manually, you may also follow these steps. Compile your interpreter by typing:

```
gcc -std=c99 src/cfeeny.c src/utils.c src/ast.c
   src/bytecode.c src/compiler.c src/vm.c
   -o cfeeny -Wno-int-to-void-pointer-cast
```

at the terminal. This will create the `cfeeny` executable.

Next, you have to run the supplied parser which will read in the source text for a Feeny program and dump it to a binary AST file.

```
./parser -i tests/cplx.feeny -oast output/cplx.ast
```

Finally, call the `cfeeny` executable with the binary AST file as its argument to start the interpreter.

```
./cfeeny output/cplx.ast
```

3 Object Layout

The simplicity of the garbage collector depends greatly upon the representation of objects and of your machine state. The first step is to refactor your bytecode interpreter to ensure the following object layout:

3.1 Heap Object Layout

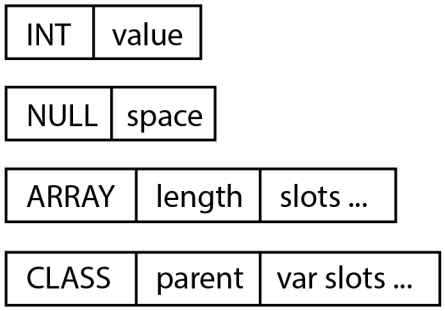

Figure 1: Heap Object Layout

An integer is represented by a 128-bit (or 2-word) piece of memory. The first word is a unique integer, INT, that indicates that the object is an integer. The second word contains the value of the integer.

The null object is represented by a 128-bit (or 2-word) piece of memory. The first word is a unique integer, NULL, that indicates that the object is the null object. The second word is reserved for use later. *Do not* represent the null object using the zero-pointer. The layout described above is needlessly general, but we will optimize the representation in a later assignment.

The array object is represented by a unique integer, ARRAY, followed by a word for storing the length of the array, followed by a word per slot in the array.

Objects are represented with an integer representing the class of the object, CLASS, followed by a word for storing the parent of the object, followed by

a word for each variable slot in the object. There is a unique CLASS integer for each declared class in the Feeny program. Note that there are no words allocated on the heap for the method slots of the object!

3.2 Global Variable Layout

Store the global variables in a single contiguous array with one word dedicated to each global variable:

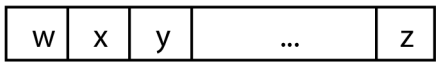

Figure 2: Global Variable Layout

Note that there are no words allocated for global functions, only global variables.

3.3 Local Frame Layout

Allocate a single array in which to consecutively store all the local frames. Each frame stores a word that contains the starting index of the parent frame, a word for storing the return address of the caller function, followed by a word for each slot in the frame.

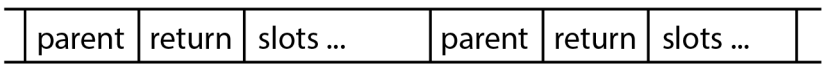

Figure 3: Frame Layout

3.4 Operand Stack Layout

Allocate a single array in which to store the values in the operand stack. Later, you may optimize your implementation to store both the operand

stack and the local frames in the same array, but keep the two separate for now.

4 Implementation Guide

4.1 Writing a Custom Allocator

Allocate a 1MB piece of memory for serving as our custom heap for the Feeny VM. Write a function:

```
void* halloc (int nbytes)
```

that returns the next piece of memory of `nbytes` bytes. Do this by keeping track of how full the heap is, and incrementing this count each time `halloc` is called.

If a call to `halloc` *will* exceed the capacity of the heap, then write and call the function:

```
void* garbage_collector ()
```

to clean up the heap before attempting the allocation. If there is still not enough space to allocate the object then print a message to signal that the system is out of memory and quit the program.

We will write the `garbage_collector` function in parts through this assignment. For now, just quit the program, and test your VM to ensure that programs work with the custom allocator until the point where the heap is exhausted. It may be useful to ensure that printf statements always flush to see the progress of the program before termination.

4.2 Scanning the Root Set

The root set of a Feeny program consists of the global variables, the stack of local frames, and the operand stack. Write a function:

```
void scan_root_set ()
```

that iterates through each variable in the root set and prints out its 64-bit address. Ensure that each address points to a region within the allocated heap. Modify `garbage_collector` to call `scan_root_set` before quitting the program.

Iterating through the global variables and the operand stack is straightforward. Finding all the variables in the local frames is trickier and involves writing a stack walker. Starting from the current frame pointer, follow the parent frame pointers to print out all the slots in the stack frames.

4.3 Copying to Free Space

At the start of `garbage_collector`, allocate another 1MB piece of memory for the free space, the region to which we will copy all the currently live objects. Use a pointer to keep track of how full the free space is.

Write a function:

```
void* get_post_gc_ptr (void* obj)
```

that, given a pointer to a heap-allocated object, returns the new pointer to the object after garbage collection.

`get_post_gc_ptr` needs to do three things:

1. The first time `get_post_gc_ptr` is called on an object it (shallow) copies the object to the free space, and returns the pointer to the new object in free space. You can determine the size of the object to be copied from the unique integer at the head of every object.
2. Before returning the new pointer, record this new pointer as the forwarding address for the old object. Do this by overwriting the contents of the old object with a "broken heart" object.

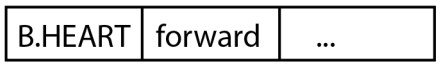

Figure 4: Broken Heart Layout

B.HEART is a unique integer indicating that it is a broken heart, and forward is the post GC pointer for the object. This overwriting is always safe to do because you've allocated a minimum of two words for every object type.

3. Subsequent calls to `get_post_gc_ptr` for an already copied object must return the cached forwarding address for the object. You can determine if an object has already been copied by checking whether it has a B.HEART tag.

The term "broken heart" was coined by David Cressey, who wrote a garbage collector for MDL, an early dialect of Lisp. (I remember the story was that the integer tag used for the forwarding object shows up as a broken heart symbol when printed to the console. But I can no longer find the source of this story.)

4.4 Copying the Roots

Now that you have `get_post_gc_ptr` written, you can use it to copy the root set of the Feeny VM. Modify your implementation of `scan_root_set` so that instead of simply printing out each variable in the root set, it replaces each variable with its post GC pointer.

4.5 Scanning the Heap

After copying the root set, all the objects directly pointed to by the root set will have been copied into the free space. However, the internal fields in those objects still point to objects in the old heap.

In `garbage_collector`, after the call to `scan_root_set`, iterate through the objects in the free space from beginning to end, replacing every internal field in each object with its post GC pointer. Use the integer tags at the beginning of every object to determine the size of the object, and the location of all of its internal fields.

You will notice that as you iterate through the free space, calls to `get_post_gc_ptr` will result in more objects being copied to the free space.

Garbage collection will end when you finally reach the last object in the free space, and it has no uncopied objects in its fields.

This step completes the garbage collector. Delete the memory used for the old heap and return from `garbage_collector` to continue running the program.

4.6 Stress Testing the Collector

The following stress test ensures that your VM is correctly written so as to allow the garbage collector to be safely runnable whenever `halloc` is called.

Currently `halloc` only calls the garbage collector when it exceeds the capacity of the current heap. Modify `halloc` to *always* call the garbage collector no matter the occupancy of the current heap, and ensure that your programs still run (albeit excruciatingly slowly) to completion.

4.7 Flipping Heap space with Free space

There is currently an unnecessary allocation of free space and deletion of heap space on every call to the garbage collector.

Modify the garbage collector so that instead of deleting the memory used for the old heap after collection, use the memory for the free space the next time the collector is called.

5 Report

Implement your bytecode compiler and interpreter (with GC) and place it in the `src` directory, naming it `compiler_xx.c` and `vm_xx.c`, where `xx` is replaced with your initials. Then ensure that it can be properly driven by the testing harness by typing:

```
./run_tests compiler_xx.c vm_xx.c
```

Include your implementation of Towers of Hanoi, Stacks, and More Towers of Hanoi from exercise 1, and ensure that your compiler and interpreter works correctly for those as well.

1. (60 points) **Compiler Statistics**: Create a table in your report that measure and calculate the following statistics for the test programs: `bsearch.feeny`, `inheritance.feeny`, `cplx.feeny`, `lists.feeny`, `vector.feeny`, `fibonacci.feeny`, `sudoku.feeny`, `sudoku2.feeny`, `hanoi.feeny`, `stacks.feeny`, `morehanoi.feeny`.

   (a) Total amount of time (in milliseconds) for `interpret_bc` to run and return.

   (b) Total amount of time (in milliseconds) spent in the `garbage_collector` function.

   (c) Total number of bytes allocated by `halloc`.

   (d) Total number of bytes allocated by `halloc` for integer objects.

2. (10 points) **Host Language Interactions**: One of the hardest aspects of programming a garbage collector is correctly handling the interaction with the host language whose variables are not visible to our root set calculator.

   Point out the error in the following pseudocode for implementing the ARRAY bytecode instruction.

```
case ARRAY_INS :
   val init = pop from operand stack
   val length = pop from operand stack
   val array = halloc(8 + 8 + 8 * length.value)
                    //Tag + Len + Slots ..
   array[0] = ARRAY_TAG
   array[1] = length.value
   array[2 to 2 + length.value] = init
   push array to operand stack
```

3. (5 points) **Bytecode vs AST Interpreter**: Imagine writing a garbage collector for our AST interpreter for Feeny. Would it be more or less difficult than writing the garbage collector for our bytecode interpreter? What specific aspects make it more or less difficult?

4. (5 points) **Long Living Objects**: Suppose for one of our programs we allocate a very large array at the beginning of the program, and read and write from it for the duration of the program. Comment on the negative performance characteristics of our current garbage collector on such a program.

5. (5 points) **Primitive Arrays**: Suppose we are writing a program dealing with heavily numerical data which operates primarily on large arrays containing integers through the duration of the program. Comment on the negative performance characteristics of our current garbage collector on such a program.

6. (5 points) **Linked Lists**: Suppose we are writing a program that makes heavy use of linked lists. The program operates primarily on many long linked lists of integers. Comment on the negative performance characteristics of our current garbage collector on such a program.

7. (5 points) **Performance Complexity**: Comment informally on the performance complexity of our garbage collector. What aspects of the program is it dependent on?

8. (5 points) **Foreign Function Interface**: Suppose we add the ability to Feeny to call functions defined in other languages (ie. C), to which we will pass out pointers to Feeny objects. Compared to an FFI for a non-garbage collected language, describe the additional complications that must be handled correctly when designing an FFI for Feeny.

6 Deliverables

Students may work in pairs or alone. Please submit your answers to section 5 as *report_XX.pdf*, and your programs as *compiler_XX.c*, *vm_XX.c*, *hanoi_XX.feeny*, *stacks_XX.feeny*, and *morehanoi_XX.feeny*. Zip all files together in a file called *assign5_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny5]` in the subject header.