Module 11: Garbage Collection

Contents

In this module you'll recycle memory that is allocated on the VM heap, so that vScheme programs can allocate far more memory than the SVM actually takes from the operating system.

- *What am I doing?*
  - Implement a garbage collector, which reuses old cons cells, closures, and other objects that are no longer needed by the computation. Your garbage collector will pack live objects densely into pages and then recover entire pages for use in future allocation requests.
- *Why am I doing it?*
  - Automatic memory management is an essential feature of almost every programming language.[1]
  - Dividing free space into large contiguous chunks ("pages") enables *extremely* fast allocation—the allocator never searches for a free block; it just grabs a chunk from a page. Fast allocation is essential for functional languages, which typically allocate new heap objects at very high rates.

    Free space is made contiguous by copying old objects out of the allocation area. As a bonus, copying garbage collection is the simplest algorithm out there.
- *How?*
  - Before lab, you'll read parts of a book chapter that explain copying garbage collection, including tricolor marking. Or if you prefer, you can watch videos. You'll also get your VM state ready for garbage collection.
  - In lab, you'll implement a scanner for your VM state, which will scan all of the state's components and will "forward" every pointer that points to a payload allocated on the VM heap.
  - At the end of the week, you'll deliver a working VM that includes a copying garbage collector. The collector will recover and reuse memory and will manage the heap's size in response to the program's demands for memory.

[1] Except for languages whose primary use case is implementing operating systems and similar low-level tasks.

The module step by step

Before lab

(1) *Download updates.* Update your git repository in two steps:

A. Be sure all your own code is committed. Confirm using `git status`.

B. Update your working tree by running `git pull` (or you might possibly need `git pull origin main`).

I'm not expecting any merge conflicts.

(2) *Extend your VM state with a pending value.* Add this new field to your `struct VMState`:

```
Value awaiting_expect; // value passed to the pending `checl
```

The `check` primitive has been updated to hold its value in this field, where the value stays until the corresponding `expect` primitive runs. The update is necessary in order to prevent the value from being garbage collected in the interval between `check` and `expect`.

Confirm that your SVM compiles.

(3) *Learn garbage-collection basics.* For a gentle, limited introduction to the basic ideas, you can watch my 5-minute video. To learn the terminology, like "live data" and "tricolor marking," I recommend using excerpts from my book chapter. Start with the reading guide.

If you want more, I've also found ~~three~~ two outside videos, all of which illustrate garbage collection with animated diagrams:

Redgate, .NET collection, 4 minutes, YouTube

Pros and cons:

- Short
- Very good job setting context
- Not exactly our algorithm
- Does not teach any of the jargon

Notes:

- .NET is a stack machine, not a register machine; their stack corresponds to our registers.
- Their "statics" correspond to our literals.
- Globals are the same.
- Their algorithm is mark/compact, sometimes called mark, "sliding" compact. This algorithm copies live objects to the same memory they already occupy; our algorithm copies live objects to fresh memory. Ours is simpler.

Matthew Flatt (University of Utah), copying collection, 5 minutes, YouTube

Pros and cons:

- Short
- Exactly our algorithm
- Good-enough animation of garbage collection
- No extraneous details
- No details

Notes:

- The blue box on left is our `current` list.
- The blue box on right is our `available` list
- The purple paint is a non-NULL `forwarded` pointer.
- Matthew Flatt is the chief implementor of the Racket system; he knows what he's about.

Christoph Reichenbach (Goethe University of Frankfurt), copying collection, 15 minutes, Lund University Canvas

Pros and cons:

- Motivates contiguous free space from fragmentation
- Best diagram/animation of the algorithm
- Good discussion of requirements circa 11:10 (slide 7/9)
- Opening will make you think you forgot to study mark and sweep
- Only approximately our algorithm
- Disingenuous or uninformed analysis

Notes:

- Copying is a species of compaction.
- Uses Cheney's algorithm: to-space also acts as a queue of gray objects (`scan` and `next` implement the queue). We have just the next pointer! What for Cheney is "between `next` and `scan`" for us is "on the `gray` list".
- Move is our `forward_*`. (And his move doesn't copy the object! "That is implicit.")
- Criticism "twice the heap size" is disingenuous (or ill-informed). Also doesn't mention pause times or locality.

None of these videos is necessary for your understanding. But they may be the easiest way to get started. If it were me, I'd spend 10 minutes on the first two videos and skip the third (except possibly to look at the diagram/animation).

(4) *Study tricolor marking and object scanning.* Read about tricolor marking in my book chapter on pages 255–266. In lab you'll do the step "color all the roots gray." (As you're reading the chapter, don't overlook the reading guide.)

To see how object scanning can be implemented in the SVM, look at function `scan_value` in file `vmheap.c`. This function, which is analogous to function `scanloc` on page 282 of the book chapter, finds a gray object's white successors and colors them gray. You'll be writing analogous functions for your `struct Activation` and `struct VMState`.

To scan your VM state successfully, you'll need to use forwarding functions. Look at two functions in file `vmheap.c`:

- Function `forward_string` forwards the heap-allocated payload of a VM string. (File `vmheap.c` includes four other functions that do the same thing for payloads of other types: functions, closures, blocks, and tables. Reading `forward_string` is enough.)
- Function `forward_payload` looks at a `Value`, and if the `Value`'s payload is allocated on the heap, it forwards the payload. It also colors the forwarded object gray (if the object's payload might contain pointers to other payloads) or black (if the object's payload cannot possibly contain a pointer to another payload).

(5) *Write macros needed to save cached state to your* `struct VMState`. As described in "The garbage collector and the VM state", some of your VM's state is likely kept in local variables of `vmrun`. And before a garbage collection, some of this state may need to be flushed to the `struct VMState` record that is allocated on the C heap. **Any pointer to data allocated on the VM heap might move** and so has to be flushed

to the cache. In addition, you must flush other data of interest to the garbage collector, like the position of the register window.

After each garbage collection, local variables of `vmrun` may need to be updated, because the VM state that they represent may include locations of objects that have moved. Local variables can be saved and then updated by two macros `VMSAVE` and `VMLOAD`[2]

My own `VMSAVE` macro saves the currently running function and the current register-window pointer. My `VMLOAD` macro restores the currently running function and also a pointer to that function's instructions, which is cached. It doesn't have to restore the register-window pointer because register windows aren't allocated on the VM heap, so they don't move.

Define `VMSAVE` and `VMLOAD` macros that are consistent with your own data structures.

(6) *Get your VM state ready for garbage collection.* Any payload that can be reached by following pointers must be reachable from your `struct VMState`. To make that possible, you may have to curate your code to establish the following invariants:

- The VM state must not contain any pointers of type `Instruction *`. Such pointers are examples of "interior pointers" and they complicate garbage collection unreasonably. They are most likely to be found on the call stack.

  This invariant can be established by replacing every instruction pointer with a combination of a `struct VMFunction *` and an integer index. (It is OK to use `Instruction *` internally in `vmrun`, just not in the VM state.)

- The VM state record must be capable of storing the currently running function and its program counter. You may have to add fields to your `struct`.

(7) *Prepare to compile with Valgrind.* File `vmheap.c` ships with macros that tell Valgrind how we are using memory. This code should compile on the department servers with no problems. If you are compiling on your own machine, you'll need to install a Valgrind package, usually called a "developer" package. If you can't get that to work, alter the `GNUmakefile` to include `-DNOVALGRIND` in the `CFLAGS` used to compile `vmheap.c`.

(8) *Familiarize yourself with the rest of the heap implementation.* The previous steps are essential to your progress in lab. This next step will just make it easier:

Using the guide to `vmheap.c`, page through file `vmheap.c` to familiarize yourself with what's there.

Lab

(9) *Scan an activation record.* In file `vmheap.c`, implement function `scan_activation`. Ideally an activation contains exactly one reference to a heap-allocated payload, which should be a reference to the function whose activation it is. That reference needs to be forwarded. If you encounter anything else, check with a member of the course staff.

(10) *Scan the SVM state for roots.* Guided by the generic description of roots on page 265 of the book chapter, as well as the comments in file `vmheap.c`, implement function `scan_vmstate` in file `vmheap.c`.

Function `scan_vmstate` shouldn't scan *all* the registers, activations, globals, or literals that are allocated in the state record. It should scan only the ones that are in use—that is, the ones whose values might affect future computations. That includes

- The registers used by the currently active function, and all lower-numbered registers
- The activations that hold suspended computations, i.e., the ones that are actually on the call stack
- The slots that have actually been allocated to hold globals or literals[3]

Scanning registers and activations that are not actually used will result in memory leaks that Valgrind can't detect.

After lab

After lab, your work will alternate between your garbage collector and your mutator. (In the perverse jargon of garbage collection, the "mutator" is the part of your program that's doing useful work—but the collector sees it only as an annoying distraction that comes in and mutates its carefully managed heap.)

Be alert to a memory-management pitfall

(11) *Check your loader for allocations.* Check your file `loader.c` for any code that might allocate space for a function. **If you allocate that space with `malloc`, there may be trouble ahead.** A `struct VMFunction *` points to the payload of a `Value` with the `VMFunction` take, and when the garbage collector sees the value, it's going to think that it owns the memory allocated to the payload. I don't think havoc will ensue,

[2] When possible, I prefer `static inline` functions over macros, but because `VMSAVE` and `VMLOAD` need access to `vmrun`'s local variables, they can't be defined as `static inline` functions.

[3] For depth points, you can avoid scanning the literals in this step and can garbage-collect unused literals.

but if you violate the heap invariants, I can't warrant the results.

Change the allocation to use `VMNEW` as described below.

(12) *Check for other calls to* `malloc`. Search the code you've written for other calls to `malloc`, `calloc`, and `realloc`. Unless you're looking at allocation of the VM state record or one of its components, these calls might need to be replaced with allocation operations from the VM heap.

To distinguish your code from mine, you might find it useful to use `git blame`. Start a `bash` instance, and try

```
for i in *.c; do git blame $i; done | egrep -w 'malloc|realloc|calloc' | grep -v 'Norman Ramsey'
```

Expect to see the allocation of the VM state record and its components, but nothing else.

Prepare your mutator

(13) *Correct direct calls to* `vmalloc_raw`. The invariants of the garbage collector require that each heap-allocated object include a `forwarding` pointer and that the `forwarding` pointer of a newly allocated object be initialized to `NULL`. You must either initialize the pointer yourself or use macro `VMNEW`. This macro takes a type, a variable name, and a number of bytes, and it both declares and initializes the variable. Here is an example from my function loader:

```
VMNEW(struct VMFunction *, fun, vmsize_fun(count + 1));
```

I recommend using the macro because then you can use `grep` as an oracle to know if you've gotten things right. Searching for `vmalloc_raw` should produce two hits in `vmheap.c` and two in `vtable.c`, and that is all:

```
$ grep vmalloc_raw *.c
vmheap.c:void *vmalloc_raw(size_t n) {
vmheap.c:  void *block = vmalloc_raw(num * size);
vtable.c:            p = vmalloc_raw(sizeof(*p));
vtable.c:   struct binding *copy = vmalloc_raw(sizeof(*p));
```

These four hits don't need to be changed. If you find any other uses of `vmalloc_raw`, change them.

(14) *Get right with* `valgrind`. Using your existing test codes, run your SVM with `valgrind` and confirm that memory is used correctly and it is all recovered.

```
vscheme -predef | uft ho-vo > predef.vo
uft ho-vo scheme105.scm | valgrind --leak-check=full svm predef.vo -
```

Depending on the nature of your tests, you may get an assertion failure complaining about "large-object allocator not implemented." If that happens, go into file `vmheap.c` where `PAGESIZE` is defined to be 600. Double that size, and keep doubling it until your code will run.

You want the output to look something like this:

```
==13551== HEAP SUMMARY:
==13551==     in use at exit: 0 bytes in 0 blocks
==13551==  total heap usage: 18,839 allocs, 14,297 frees, 1,
==13551==
==13551==   All   heap   blocks   were   freed   --
no leaks are possible
```

If you do find leaks, one possible source is that you might be using `malloc` to allocate things like closures, cons cells, or functions. Use the `VMNEW` macro instead. There should be no calls to `malloc` in `vmrun.c` and only one in `loader.c`; confirm by

```
fgrep -w malloc vmrun.c loader.c
```

Note: This step is here because Valgrind can be a great tool to help you find bugs in your garbage collector—but only if it isn't distracted by bugs somewhere else. If you find yourself stuck (or slow) on this step, get help.

Build your collector

(15) *Familiarize yourself with the heap implementation.* If you skipped step (8), or if you just want to refresh your memory, use the guide to `vmheap.c` to familiarize yourself with what's there.

(16) *Implement the garbage collector.* Using your `scan_vmstate` from step (10), implement the `gc` function in file `vmheap.c`. The main part of the algorithm looks roughly like this:

- Grab all the allocated pages (including the `current` pages) and keep a pointer to them (in a local variable). These are your from-space pages. Your to-space pages are in `available`.
- Color all the roots gray. By design, it suffices to scan your `struct VMState`.
- As long as there is a gray object, remove it from the `gray` stack and scan it. If it has any white successors, the `scan_value` function will make them gray (or black). (The interface in file `vstack.h` will be helpful here.)
- Call function `VMString_drop_dead_strings()`. This function removes dead strings from a persistent data structure.
- Using function `make_available`, move the from-space pages to the `available` list. This is the "flip" described in the book.

As functions `scan_vmstate` and `scan_value` copy objects, the allocator will migrate to-space pages from the `available` list into the `current` list. This is OK.

In addition to the main part of the algorithm, the `gc` function does a few other things:

- Before starting to scan and copy objects, it sets flag `gc_in_progress`. This flag ensures that all copy operations are counted as copies, not as allocation requests. When the collection is complete, function `gc` must clear the flag.
- Before returning control to the mutator, `gc` looks at the `count.current.pages`, the number of pages in use. This counter measures the amount of live data. In case the heap needs to be enlarged, `gc` calls function `growheap` with a value obtained from `target_gamma`.

Get your SVM ready to find bugs

Copying garbage collection uses simple algorithms and data structures. But the engineering requires a lot of attention to detail, and overlooking an important pointer is all too easy. To armor yourself against such oversights, I recommend altering your SVM.

(17) *Add a GC instruction and primitive (optional).* I recommend defining a new instruction with opcode `gc` whose only action is to run the garbage collector. To make it useful, you'll also have to add `gc` to your UFT in file `primitives.sml`: a side-effecting primitive with arity zero. Using this primitive, you'll be able to trigger a garbage collection whenever you want—perhaps before every `cons`, as in this example:

```
(let ([cons (lambda (x xs) (begin (gc) (cons x xs)))]) ...)
```

Implement the `gc` SVM instruction and vScheme primitive.

(18) *Add instrumentation to* `vmrun.c`. If your code accidentally refers to an old version of an object, the old version's `forwarding` pointer will be non-NULL. A reference might be caught by `valgrind`, but such references can also be checked by using the `GCVALIDATE` macro defined in file `gcmeta.h`. This macro wraps a reference to a heap-allocated payload by adding an assertion. The reference is duplicated, so use it only with a named variable or a reference to a field of a named `Value`.[4]

The `GCVALIDATE` macro is used automatically whenever you use a macro like `AS_CONS_CELL` or `AS_VMSTRING`. But your implementations of `call` and `tailcall` might refer to a `struct VMFunction` pointer directly, and wrapping these references in `GCVALIDATE` is a good idea.

Ensure that every reference to a heap-allocated payload is wrapped in the `GCVALIDATE` macro.

Run and test your collector

(19) *Enable garbage collection in* `vmrun`. The need for a garbage collection is established by function `newpage` (called from `alloc_small` via `take_available_page`) when the number of pages on the `available` list drops below `gc_when_available_at_most`. This event occurs far away from `vmrun`, where it isn't yet known that space has run out. So `newpage` cannot safely call `gc` directly. Instead, it sets the flag `gc_needed`, which signals to `vmrun` that it should initiate a garbage collection when it is safe to do so. In this step, we teach `vmrun` to respond to the signal.

Initiation of a garbage collection is complicated by a desire for efficiency. We don't want to pay the cost of checking `gc_needed` on every garbage collection. Fortunately such frequent checks are not necessary. It is sufficient to check *at every call* and *at every backward branch.* Because every loop must be interrupted by a call or a backward branch, checking at these places ensures that the mutator demands only a small, finite amount of space before initiating a collection.

Update your `vmrun` function so that on every ordinary call, every tail call, and every backward branch,[5] it checks `gc_needed`. If `gc_needed` is set, `vmrun` should save its local state to the VM state record, call `gc` with that record, and finally restored its local state from the VM state record. To perform this operation, I've defined this macro:

```
#define GC() (VMSAVE(), gc(vm), VMLOAD())
```

If you've defined a `gc` instruction in step (17), be sure to use `GC()` (or `VMSAVE` and `VMLOAD`) there as well.

(20) *Preliminary tests of your collector.* Try out your collector with a couple of simple codes that allocate `cons` cells and very little else.

Allocate and discard a thousand cons cells

```
; file alloc.scm
(define allocate (N)
  (let ([x #f])
    (begin
      (while (> N 0)
             (begin
               (set x (cons 'a 'b))
               (set N (- N 1))))
      x)))
```

[4] It's a macro, not an inline function, because it's meant to simulate polymorphism.

[5] If, as I recommend, you've implemented PC-relative branches, you can identify a backward branch by its negative offset. If your `goto` uses absolute offsets, you will have to compare the target offset with the current program counter.

```
(check-expect (allocate 1000) (cons 'a 'b))
```

Allocate a half-million cons cells

Only 1,000 cons cells are live at any one time:

```
; file grow.scm
(define iota (N)
  (let ([ns '()])
    (begin
      (while (> N 0)
             (begin
               (set ns (cons N ns))
               (set N (- N 1))))
      ns)))

(check-expect (iota 8) '(1 2 3 4 5 6 7 8))

(define allocate (N)
  (let ([x #f])
    (begin
      (while (> N 0)
             (begin
               (set x (iota N))
               (set N (- N 1))))
      x)))

(check-expect (allocate 1000) '(1))
```

Using my collector, some sample runs look like this:

```
$ uft ho-vo alloc.scm | env SVMDEBUG=gcstats svm
The only test passed.
Requested 49,056 bytes in 1,013 allocations
10 garbage collections copied 9,552 bytes
The collector copied 0.19 bytes for every byte requested
At exit, heap contained 1 used pages and 2 available pages
Total heap size is 18,000 bytes held in 3 pages

$ uft ho-vo grow.scm | env SVMDEBUG=gcstats svm
All 3 tests passed.
Requested 24,028,976 bytes in 500,591 allocations
546 garbage collections copied 27,293,520 bytes
The collector copied 1.14 bytes for every byte requested
At exit, heap contained 11 used pages and 23 available pages
Total heap size is 204,000 bytes held in 34 pages
```

Until you implement step (22), **your results may look different.**

(21) *Test with Valgrind.* Run the same tests with `valgrind --leak-check=full`.

Now go to `vmheap.c` and enlarge the `PAGESIZE` from 600 bytes to 6000 bytes.

Recompile, then run your `scheme105.scm` test from module 9, again with `valgrind --leak-check=full`.

Manage the size of the heap

(22) *Implement the heap-growth policy.* If a program starts to run short of memory, it can start to spend all its time trying to garbage-collect what little memory it has left. Our page-based allocator militates against this problem, but to prevent it completely we need a policy: *when memory runs short, grow the heap.*

I recommend a simple heap-growth policy with one parameter called γ ("gamma"). The parameter measures the ratio of the heap size to the amount of live data. The heap-growth policy sets a minimum acceptable γ, which is obtainable by calling function `target_gamma` defined in file `vmheap.c`.

The policy parameter is taken from vScheme global variable `&gamma`. When that variable is not a number, a default is used. Useful values range between, say, 2 and 10. (If `&gamma` is over 100, `target_gamma` interprets it as a percentage, so if you try setting `&gamma` to 500, that's a heap 5 times the size of live data.)

In file `vmheap.c`, alter function `growheap` to implement the policy: if the total heap size is not at least live data times `target_gamma`, add pages until it is. When the heap grows, announce it:

```
if (grew && svmdebug_value("growheap"))
  fprintf(stderr, "Grew heap to %d pages\n",
            count.current.pages + count.available.pages);
```

(23) *Final tests before submission.* In addition to the tests from step (20), run some tests that allocate and discard a lot of memory. Good candidates include Quicksort and merge sort on large lists. (Insertion sort is good only if the list is not already sorted or nearly sorted.) Try them and be sure the heap grows as you expect.

What and how to submit

(24) On Monday, *submit the homework.* In the `src/svm` directory you'll find a file `SUBMIT.11`. That file needs to be edited to answer the same questions you answer every week. To submit, you'll need to copy your working tree to the department servers. We recommend using `rsync`, but `scp` also works.

Now log into a department server, change to your working tree, and submit your entire `src` directory:

```
provide cs106 hw11 src
```

or if you keep an additional `tests` directory,

```
provide cs106 hw11 src tests
```

(25) On Tuesday, *submit* your reflection. Create a plain text file `REFLECTION`, which will hold your claims for project points and depth points.

For each project point you claim, write the number of the point, plus whatever is called for in the section "How to claim the project points"—usually a few sentences.

Now copy your `REFLECTION` file to a department server and submit it using `provide`:

```
provide cs106 reflection11 REFLECTION
```

Learning outcomes

Outcomes available for points

You can claim a project point for each of the learning outcomes listed here. Instructions about how to claim each point are found below.

1. *Applications.* You understand applications of garbage collection.
2. *Memory management.* You can write a C program with no memory errors and no leaks.
3. *Roots.* You understand roots.
4. *Forwarding-pointer invariants.* You can show where forwarding-pointer invariants are established for new objects.
5. *Color invariants.* You can explain the roles of color invariants.
6. *Cached state.* You understand what cached state is flushed at garbage-collection time, and why.
7. *Forwarding, part 1.* You understand what information is needed to forward a payload pointer.
8. *Forwarding, part 2.* You understand why we check a forwarding pointer before copying an object.
9. *Heap-growth policy.* You understand how the heap-growth policy parameter affects space-time tradeoffs.
10. *Structure of the heap graph.* You can explain how copying collection preserves linked pointer structures.

You can claim depth points for improving your SVM:

11. *Denser cons cells [2 points].* Using a `struct VMBlock` to represent a cons cell has its advantages, but the machine word spent on `nslots` is wasted. Define a new type of payload that carries just two values, `car` and `cdr`, plus GC metadata. Make your system use that payload for cons cells.

    Measure the improvements in heap size and bytes requested.

12. *Densest cons cells [2 points].* The previous depth opportunity reduces the size of a cons cell from 48 bytes to 40 bytes. But the 8 bytes spend on a forwarding pointer are used only during garbage collection, and are otherwise wasted. Moreover, once the cons cell has been forwarded, the `car` is no longer needed. Alter the representation of `cons` cells so that the forwarding pointer and the `car` share space. (An anonymous union will be helpful here.) This will reduce the size of a cons cell to its absolute minimum of two values (32 bytes).

    Measure the improvements in heap size and bytes requested.

13. *Large objects [3 points].* Implement a large-object allocator. A large object should be allocated with `malloc`, not on a page, and should never be copied. Here are some tricks:

    - You can identify a live large object by setting its `forwarded` pointer to point to itself.
    - You can link large objects on a list, much as pages are linked on a list or interned strings are linked on a list.
    - After a collection, you can reclaim dead large objects by traversing the list after the manner of function `VMString_drop_dead_strings`.

    To test your allocator, you'll want to allocate large objects and let them die. Try loading large functions into global variables, then set those global variables to `nil`, allowing the large functions to die.

14. *Interior pointers [4 points].* Make it possible to keep a function alive (and copy it to a new location) even if the only reference to the function is an interior pointer to its instruction stream (of type `Instruction *`). And also to correctly forward such interior pointers.

    The key operation here is to be able to take a pointer to an arbitrary location in an instruction stream and somehow to find the function to which that instruction stream belongs. Perhaps the operation can be implemented via some clever doctoring of the instruction stream.

    This little project might turn out to be more than 4 points worth of work, but it's less than 4 points worth of interest, so I'm just going to keep it at 4 points. Nobody loves interior pointers.

15. *Zero fewer registers [2 points].* At each VM call, update a high-water mark that records the highest-numbered register ever used. At each garbage collection, zero registers only to the high-water mark, then reset the high-water mark. Using a memory-intensive benchmark, measure the difference in both run time and number of hardware instructions executed.

16. *Garbage-collect registers based on liveness analysis [5 points].* Either in the UFT or the SVM, do a static analysis at each GC safe point to determine which VM

registers are actually live. At GC time, consult the results of the analysis to know exactly which registers to scan.

How to claim the project points

Each of the numbered learning outcomes 1 to 10 is worth one point, and the points can be claimed as follows:

1. It is convenient to be able to use `new` without `delete` or `malloc` without `free`. But it is even more convenient to write code without thinking about allocation at all. To claim this point, give an example from a language not in the Scheme family, where the implementation of a feature just allocates behind the scenes, and the garbage collector takes care of the rest. To earn the point, it must be a feature that a programmer can use without being forced to think, "I am allocating memory here."

2. To claim this point, get your SVM into shape where you can run it on a program and Valgrind reports no errors and no leaks. (You are welcome to do this in step (14), *before* you start your garbage collector.) Commit the code and tag it with

   ```
   git tag valgrind-clean
   ```

   Submit the output from `valgrind` running your SVM on the program of your choice.

3. To claim this point, identify the lines in your `scan_vmstate` function that scan registers and literals, and explain how you know *which* registers and literals to scan.

4. To claim this point, look at the lines in your source code that initialize the payload for a `ConsCell`, and identify the line that initializes the payload's `forwarded` pointer to NULL.

5. If a value that is gray or black were accidentally put on the `gray` list a second time, this would violate one of the color invariants. To claim this point, explain what bad thing could happen if this invariant is violated.

6. In step (19), I observe that my `VMSAVE` macro flushes the current register-window pointer. But my `VMLOAD` does not reload it. To claim this point, explain why I would bother to save this value if I'm not reloading it.

7. To claim this point, explain why it's not possible to forward a pointer of type `Instruction *`.

8. Function `forward_string` copies a payload only if the forwarding pointer is `NULL`. To claim this point, suppose instead that `forward_string` copies a payload unconditionally every time it is called, and explain what could go wrong.

9. To claim this point, say how you would set the heap-growth policy parameter to do less work by using a lot of space. Then say how you would change the policy to use less space at the cost of doing more work. If possible, support your answer with evidence of `gcstats` output from your collector. If not, explain why it is not possible.

10. In file `value.c`, function `eqvalue` compares two `String` values as equal only if they point to the same payload—if two `Values` point to different payloads they are considered different strings, even if the payloads contain the same characters.

    Suppose that a string value appears in two different VM registers, so both registers point to the same payload. But then the garbage collector copies every register's payload to a new location. To claim this point, explain how the collection manages to preserve pointer relationships so that after the collection, the registers are still considered to hold equal strings.