PLC: Homework 7 [150 points, 165 max]

Deadlines

All solutions are due May 9th, the last day of class.

Late homework is **not** accepted, but we will use second-chance grading as for previous homeworks. Do not forget to check your grade for the first-chance submission. We will post grades by Monday, May 12th, at noon, at the latest (but likely earlier than that). Your second-chance submission is then due Tuesday, May 13th, by noon. You are under no obligation to submit a second-chance submission, if you are satisfied with your first-chance submission. The first-chance grade will be used again for the second-chance grade, as in previous assignments.

Subversion, Partners

The same instructions as for previous homeworks apply for turning in your solution via subversion, and for extra files you need to submit if you are working with a single partner.

For this assignment, you will check in files to the `hw7` subdirectory of your personal repository (see hw1 instructions for how to check out this repository).

Simulating Automatic Memory Management with `gclang`

The goal of this assignment is to implement a simulator called `gclang` for simple automatic memory management schemes in Agda: reference counting, mark-and-sweep garbage collection, and copying collection. The simulator will read in a file containing a sequence of commands describing how the reference graph is being changed, and requesting when automatic memory reclamation should take place. Depending on command-line options, the simulator will perform one of those automatic memory management algorithms. The results of the simulation will be dumped to files describing the reference graphs as they are updated by memory management or the other commands.

I am giving you the grammar for `gclang`, as well as a starting version of `my-gclang-main.agda` which contains some helper functions for printing out heaps. You will be modifying `my-gclang-main.agda` and submitting that file (only) for this assignment.

We will work with a simplified view of memory for this assignment. A `mem` (already defined in `my-gclang-main.agda`) is a pair of a list of root locations, and a list of cells. The locations are just natural numbers indexing the list of cells. That list of cells models the heap. Each location refers to a cell in the list. Each cell has two fields, `a` and `b`, plus one extra slot (called `extra`) for holding data used by the memory-management algorithm. Initially, all components of the cell are `null`, which is modeled as `nothing` in `my-gclang-main.agda`. Fields can be modified to store a location (i.e., an index into the list of cells modeling the heap). The first address in the heap is address 0. The `extra` slot can be modified to hold whatever value a particular memory management scheme requires.

Updates to my-gclang-main.agda Sunday, May 4

I have made a few small changes to the `my-gclang-main.agda` file I am providing you, as of Sunday, May 4. At the top of the file, we now have this import (for an implementation of `merge-sort` I wrote recently):

```
open import list-merge-sort ℕ _<_
```

Also, I added an extra component to the definition of `mem`:

```
mem : Set
mem = maybe ℕ × 𝕃 ℕ × 𝕃 cell
```

This is for the benefit of the copying collection algorithm problem below. It will be a little irritating to modify code you might have already written to incorporate this field, but it should not be hard.

This is now called in `mem-to-string` to sort the roots; that function also now prints the global extra field:

```
mem-to-string : mem → string
mem-to-string (global-extra , roots , cells) = "global extra: " ^ (field-to-string global-extra
        ^ "\nroots: " ^ 𝕃-to-string ℕ-to-string " " (merge-sort roots) ^ "\n" ^ string-concat
```

Finally, the default is not to use any memory management scheme, so the very last line of the file has been changed to

```
main = getArgs >>= processArgs ff ff no-mem-management
```

You should be able to incorporate those changes into your `my-gclang-main.agda`, while retaining any code you already added to that file.

Input Files to `gclang`

The `gclang` tool takes in input files in the format described by the `gclang.gr` grammar I am supplying you (you will not need to modify the grammar for this assignment). Here is an example input file (this is `prog1.gclang` in the `hw7` directory):

```
heapsize 10

addroot 0

0.b = 1
1.b = 2
2.b = 3
3.b = 4

5.b = 6
6.b = 7
7.b = 8
8.b = 9
```

```
snapshot
```

All input files must begin with the `heapsize X` directive, where `X` is a number. This directive tells `gclang` how big a heap (the memory for cells in the reference graph) to allocate.

After the `heapsize` directive comes a sequence of commands of the following kinds:

- `x.a = y` or `x.b = y`. These say to update field `a` or `b`, respectively, of the cell at location `x` so that it stores value `y`. The location `x` is just a number. The value `y` can be a location (so also a number), or else `null`, to indicate that the field should be set (back) to null.
- `addroot x` or `droproot x`. These commands say to add or remove, respectively, the location `x` from the list of root locations which is the first component of a `mem` value (in `my-gclang-main.agda`).
- `gc`. This command requests garbage collection. It should be ignored when performing reference counting (discussed below).
- `snapshot`. This command requests that `gclang` remember a snapshot of the heap as it currently stands at that point in processing the input file. I already have some code in `my-gclang-main.agda` to print out all those snapshotted heaps to separate output files.

If a location is used, either in the `x.a = y`, `x.b = y`, `addroot x`, or `droproot x` commands, which is greater than or equal to the heap size, then those commands should have no effect.

1 Running `gclang` programs without memory management [50 points]

The first problem in this assignment is to implement support for running `gclang` programs support for the `gc` command. Example such programs are `prog1.gclang` (quoted above) and also `prog2.gclang`. To add support for running these programs, you should add code to the `process-start` function of `my-gclang-main.agda`. For a program with heapsize N, you will start of your simulation of a `gclang` program with the empty memory, which has an empty list of roots and a list of N cells that look like

```
(nothing , nothing , nothing)
```

Then you will use some helper functions, which you have to write, that can take in a `mem` value and carry out one of the commands listed above. You do not need to implement the `gc` command for this problemm; it will be implemented in later problems. You do need to implement `snapshot`, though: every time a `snapshot` command is executed, your `process-start` function should add the current `mem` value to a list of memories. Then `process-start` should return the list of all the memories that have been snapshotted. Note that the list should have the earlier snapshots earlier in the list, and the later snapshots later. This is important for how grading.

I already have code in place in `my-gclang-main.agda` to dump out the snapshots of memory in a particular format, into files called `memX.txt`, where `X` is the number of the snapshot, where 0 is for the first snapshot, and the number is incremented for later ones.

You will get 15 points for correct execution of `prog1.gclang`, and 20 for `prog2.gclang`. 15 points will be for additional hidden testcases. To confirm your solution is working, you should see a file `mem0.txt` created for `prog1.gclang`, with a description of the memory at the end of executing the program; and `mem0.txt` and `mem1.txt` files created for `prog2.gclang`. Just inspect those files to confirm that your memory is laid out as it should be following execution of those programs.

2 Implementing Mark-And-Sweep [30 points]

I already have code in place in `my-gclang-main.agda` to recognize command-line arguments for the three memory-management algorithms. For this problem, implement support for mark-and-sweep collection (command-line argument `--mark-and-sweep`) whenever a `gc` command is executed. You should use the `extra` field of cells to hold the mark bit of your mark-and-sweep implementation. When sweeping up the dead cells, you should set all their fields to `nothing`. Even though this is not necessarily exactly what would be done in a real implementation, it is how we will confirm that garbage collection has taken place. You will get 10 points each for `prog1a.gclang` and `prog2a.gclang`, which are just versions of the similarly named files, but with calls to the garbage collector. 10 points are for additional hidden tests.

3 Implementing Reference Counting [30 points]

Similarly to the previous problem, implement support for reference counting (the `--ref-counting` command-line flag). You will want to use the `extra` field to hold the reference count of a cell. If a root is added to refer to a cell, then that cell's reference count should be incremented. Also, if a field of a cell is updated to refer to a cell, then that cell's reference count should be incremented. Similarly, if we drop roots or remove field references, the reference count of the cell involved should be decremented. If decrementing a reference count makes it become 0, then that cell should be garbage-collected by setting all its fields to `nothing`. Before that, though, you have to decrement the reference counts of any cells pointed to by those fields (if they are not `nothing`). You will actually just ignore the `gc` command for reference-counting, as you will perform reference-counting operations every time a field or root is updated.

4 Implementing Copying Collection [30 points]

Similarly to the previous problems, in this problem you will implement support for copying garbage collection, which we saw in class May 1. When the user requests a heap of size X, your code should allocate one of size $2 * X$, if we are in copying-collection mode (the `--copying` command-line flag). Because `gclang` programs reference absolute locations in memory, we need to know in which half of this doubly sized heap to find a particular address. If the from-space is being stored in addresses 0 through $X - 1$, then address A just should refer to the A'th element of the list of cells comprising the heap, as for the previous problems. But if the from-space is currently being stored in addresses X through $2 * X - 1$, then your implementation should make address A refer to the element at index

$A + X$ of the list of cells. You can use the new first component of `mem` values (added Sunday, May 4; see the Update section above) to help you implement this address-translation step when reading or writing locations.

There are a few details we have to pin down for the copying step of copying collection. First, you can use the extra field of every cell to store the forwarding pointer into the to-space. After the copy is complete, this field should be set back to `nothing`. Second, I am stipulating that you should copy cells in a depth-first fashion, with field a being copied before field b. I am hoping depth-first is easier to implement than breadth-first (the algorithm in class works breadth-first), since you code it in a direct way recursively. You do need to pack cells into the first part of the to-space, as we saw in class. For this, your code for copying cells over will likely need to return an updated heap (with new cells copied into the to-space), together with the location of the next cell to copy to.

You will get 20 points for correctly handling `prog3.gclang`, and 10 more for hidden tests.

5 Potpourri

Here are several other problems you could implement for further points. We will cap your score at 165.

5.1 A Property of Reference Counting [10 points]

In a file called `verified.agda`, for 10 points state the following theorem (you do not have to prove it, just formalize the statement of the theorem itself in Agda): "after running any `cmds` object from the starting memory in reference counting mode, if a cell is reachable from the roots then its reference count is not 0".

5.2 Verifying Reference Counting [10 points]

Prove the theorem you stated in the previous problem [likely quite challenging].

5.3 Dumping to GraphViz [15 points]

In the `my-gclang-main.agda` file I provided you, there is a function called `mem-to-graphviz`, which is called to dump a representation of a memory to a GraphViz (`.gv`) file. Currently, this function does not do anything. Write code there to convert a `mem` value to a string describing the memory as a graph in GraphViz format. You should draw edges in the graph corresponding to the `a` and `b` links in cells. You should draw incoming arrows to cells to represent roots. It is probably ok to ignore the global extra field of `mem` values.

If you solve this problem, please check in a file called `graphviz.txt` as part of your solution, so that we know to check your results. You can add any notes there about the graphs you are generating. You will not get any points if your generated `.gv` files cannot be drawn by the `dot` GraphViz program. Otherwise, we will give you full credit if it looks like you can successfully generate the expected graphs (but because we have to inspect this visually, we will likely accept solutions if they appear mostly correct, and reject solutions that appear mostly incorrect).

For help on the GraphViz format, see online sources or just inspect the `.gv` files that `gratr` produces every time it runs.