Assignment 8: Storage Management

15-312: Foundations of Programming Languages
Matthew Moore (`mlmlm@cmu.edu`)

Out: Tuesday, November 16, 2004
Checkpoint: Thursday, December 2nd, 2004 (11:59 pm)
Due: Thursday, December 9th, 2004 (11:59 pm)

150 points + 30 Extra Credit

1 Introduction

In this assignment, you will implement two garbage collectors for an abstract machine. Your garbage collector will automatically manage the memory used to store pairs, lists, and closures. In the assignment directory, you'll find several files with support code; you will only need to fill in the missing code in `eval.sml`, `mark-gc.sml`, `semi-gc.sml`, `qsort.mml`, and `isort.mml`. You will also write some sample MinML programs and measure the performance of your garbage collectors on these programs.

Please note that while we allow you to implement each of the collectors as you choose, the interface for the collector must remain the same. Otherwise, our test programs will not run and you will lose a significant number of points.

You will rarely, if ever, need to write long or complicated functions to complete this assignment. Therefore, you should strive for elegance. Your solution will be graded primarily on correctness, but if your code does not correctly handle one or more cases, we will inspect your code and attempt to give you some credit for the understanding it reflects. It is therefore to your benefit to write clean, legible code.

We will be considering a similar MinML language to what we used in Assignment 4, which will include pairs and lists, but some of the details of the implementation have changed significantly. Before you begin, you may wish to read the provided code (especially the signatures) to gain an understanding of the setup. All of the necessary SML files are listed in the `sources.cm` file, and you can build the project in SML/NJ by typing `CM.make()`.

2 Checkpoint

For the checkpoint (Thursday, December 2), you must submit a working implementation of the two garbage collectors. The reason for the checkpoint is that this is a larger assignment than the previous ones. We also want to give those of you who can't complete the all tasks involved in the checkpoint a chance to complete the measurement component of the assignment. We will be posting a sample implementation to the course website after the checkpoint is due. If you couldn't get your implementation to work, then you can instrument our sample implementation for your measurements and analysis.

3 Background

One of the goals of garbage collection is to make memory management opaque to the programmer; consequently, we won't be adding any new typing rules. We will be changing the behavior of machine, however, and so we will have some new transition rules. We will also add a new component to our abstract machine, a heap H. Our evaluation judgments will now look something like:

$$H \;;\; k > e$$

We might read this judgment as "we are currently evaluating expression e with stack k and heap H."

Our extended abstract machine semantics will share some features with the structures we used to reason about references and mutable storage, and in particular, the heap will map locations l to values. It is important to understand that, unlike our description of references, the heap is not mutable: there is no way for the program to "update" the value associated with a reference. In fact, programs will not, in general, even be aware of the heap.

One of our goals in designing the E machine was to build a somewhat more realistic model of how programs are executed on real hardware. While we carefully described the interactions of the stack and the environment, we neglected to note that the physical memory used to maintain these structures can not store data of arbitrary structure or size. For example, the machine registers mostly likely used to store elements of the current environment are limited to 32 (or 64) bits, far too small to hold a nested pair structure. So even without adding references to our language, we have a need to perform memory management.

Like any good hardware (or pseudo-hardware) the E machine has done its job, but now it's time to go out and get one of the latest models,

The A machine

The A machine will provide the necessary extensions to support our implementation of an automatic memory manager or *garbage collector*. In particular, it will distinguish between *small* values (those suitable to live in the environment or on the stack) and *large* values, as shown below.

$$\begin{array}{llcl} \textit{(small values)} & v & ::= & \texttt{num}(i) \mid \texttt{true} \mid \texttt{false} \mid \texttt{unitel} \mid \texttt{loc}(l) \mid \texttt{nil} \\ \textit{(large values)} & w & ::= & \langle\!\langle \eta; e \rangle\!\rangle \mid \texttt{Cons}(v_1, v_2) \mid \texttt{Pair}(v_1, v_2) \end{array}$$

Following our comments above, the A machine will also define a *heap*, a finite map from locations to large values (also note that environments now map variables to *small* values).

$$\textit{(heaps)} \quad H \quad ::= \quad \cdot \mid H, l{=}w$$

Given a heap, the problem of garbage collection may then be phrased as, "when is it safe to remove a mapping from the heap?" Most garbage collectors will answer this question using a technique known as *tracing*. These collectors determine what is and is not garbage by following the *reachability* graph of the current state of the machine.

4 Evaluation

To become familiar with our new abstract machine, you will first complete the implementation of its transition rules. Most of the rules carry over from the E machine, with the addition of the heap to each state. Many rules have already been implemented; you will be responsible for the cases involving pairs, lists, and functions.

As it turns out, it was no accident that we asked you to think about building more efficient closures in Assignment 4: it will be a key to good performance in the A machine.

Task 1: A machine Evaluation (10 points)
Modify `eval.sml` to complete the implementation of evaluation for the A machine. In particular, you should implement those rules that appear below. (The rules for recursive bindings have already been implemented.) Note that even though we have not begun our implementation of the garbage collector, you will still use the functions `alloc` and `read`, as defined in the `GC` signature, to allocate space for large values and to lookup those values once they have been stored in the heap. Finally, you should use the provided function `closure` to ensure that your machine builds the smallest possible closures. This is important closures with unnecessary pointers into the heap will prevent many heap cells from being garbage collected, leading to very poor performance.

$$H \;;\; k \triangleright \texttt{Cons}(v_1, \Box) < v_2 \mapsto_{\mathsf{a}} (H, l = \texttt{Cons}(v_1, v_2)) \;;\; k < \texttt{loc}(l)$$

$$H \;;\; k > \texttt{nil} \mapsto_{\mathsf{a}} H \;;\; k < \texttt{nil}$$

$$H \;;\; k \triangleright \texttt{case}(\Box, e_1, h.t.e_2) < \texttt{nil} \mapsto_{\mathsf{a}} H \;;\; k > e_1$$

$$\frac{H = (H_1, l = \texttt{Cons}(v_1, v_2), H_2)}{H \;;\; k \triangleright \texttt{case}(\Box, e_1, h.t.e_2) < \texttt{loc}(l) \mapsto_{\mathsf{a}} H \;;\; k \triangleright (h = v_1, t = v_2) > e_2}$$

$$\frac{\eta' = \eta(k)|FV(\texttt{fn}(x.e))}{H \;;\; k > \texttt{fn}(x.e) \mapsto_{\mathsf{a}} (H, l = \langle\!\langle \eta'; \texttt{fn}(x.e) \rangle\!\rangle) \;;\; k < \texttt{loc}(l)}$$

$$\frac{H = (H_1, l = \langle\!\langle \eta'; \texttt{fn}(x.e) \rangle\!\rangle, H_2)}{H \;;\; k \triangleright \texttt{apply(loc}(l), \Box\texttt{)} < v_2 \mapsto_{\mathsf{a}} H \;;\; k \triangleright (\eta', x = v_2) > e}$$

$$H \;;\; k > \texttt{rec}(x.e) \mapsto_{\mathsf{a}} (H, l = \langle\!\langle \eta(k); \texttt{rec}(x.e) \rangle\!\rangle) \;;\; k \triangleright (x \overset{*}{=} \texttt{loc}(l)) > e$$

$$\frac{\eta(k) = (\eta_1, x \overset{*}{=} \texttt{loc}(l), \eta_2) \quad H = (H_1, l = \langle\!\langle \eta'; e \rangle\!\rangle, H_2)}{H \;;\; k > x \mapsto_{\mathsf{a}} H \;;\; k \triangleright \eta' > e}$$

Where the notation: $\eta(k)|FV(e)$ means the current environment restricted to the free variables of e. Another new notation worth commenting on is how we only add new bindings to the stack, ie. in the `case` rule for `Cons`, we only add the bindings h, t to the stack. This avoids excessive copying of the current environment and will improve the performance of your garbage collector (it will have to trace through much less).

Evaluation is invoked in a manner that differs slightly from previous assignments. You must now initialize the evaluator with the size of the heap, for example by typing either

```
Top.file_eval heapSize "test_file.mml";
```

at the SML/NJ prompt. If you don't specify a sufficiently large heap size, then your implementation will quit on an `OutOfMemory` exception. Once you have completed this portion of the assignment, you can test it using the `NoGC`, which is a "collector" that allocates memory, but never frees any. Be sure you provide a sufficiently large heap size when testing your implementation on the trivial collector.

5 Semi-Space Collection

A semi-space collector (perhaps unsurprisingly) divides memory into two halves, and offers exactly one half to the user's program to be used as storage.[1] If we try to perform an allocation and our unreserved space is full, we invoke the collector to try and free some space, but otherwise, we take the next available cell. We have provided you with some suggested starter code for the Semi-Space heap:

```
val heap :  HeapElement Array.array ref
val upper :  int ref
val lower :  int ref
val next :  int ref
```

The `heap` corresponds to physical memory, `upper` and `lower` correspond to upper and lower bounds on available memory, and `next` stores the index of the next available memory cell. Each slot of `heap` has type: `HeapElement`, which has three branches. `Unused` corresponds to an empty cell, `Cell` corresponds to an occupied cell, and `Fwd` corresponds to a cell that has been copied to another index (the new index is what is stored in the `Fwd` cell). The type `location` is given as `int ref` for the Semi-Space collector because when we collect, the location of a given large value in memory will change, so when we copy it to its new index in memory, we change the index that the location stores.

Task 2: Semi-Space Allocation (10 points)
Implement the `alloc` function, which should call the collector if `next` is not in the range: `[lower, upper)`. Otherwise, it should return the next available cell (updating `next` accordingly). If the collector doesn't free any memory, then `alloc` should raise the exception `OutOfMemory`. You should also implement the `read` function, which should lookup the supplied location in memory and return the large value stored there.

When we run out of memory to allocate in `alloc`, we must try and "collect" the cells which are no longer reachable. In order to determine the reachable locations, the collector must be able to discover which 'large' values are currently available to the program, and then find which other large values are reachable from those. We can find all of the large values which are available by traversing the stack and looking inside the frames and environments (a large value will appear in these places as a small value of the form `Loc`(l)). However, large values can reference other large values, so we must traverse the large values associated with every location we find as well.

Task 3: Tracing/Copying Large Values (20 points)
In the file `semi-gc.sml`, you will find several stubbed out (mutually recursive) functions: `copyStack`, `copyEnv`, `copyPtr`, `copysVal`, and `copylVal`. These functions will work together

[1]What we have been calling "user program" is often referred to as "the mutator," and we will sometimes use that terminology as well (despite the fact that, in our language, no mutation can occur).

to traverse the current stack, k, the large value we are trying to allocate space for and copy all the reachable locations into the reserved space.

Task 4: Collection (10 points)
In Semi-Space collection, what we will do is copy all of the reachable locations in our unreserved space into our reserved space by tracing through the stack, and the value we are trying to allocate space for. This is taken care of using: `copyStack` and `copylVal`. However, after all of the reachable locations have been copied into the reserved space, we must switch the roles of our to spaces. (So now the reserved space becomes the unreserved space, and vice-versa). Implement a function, which you call when you run out of memory, which calls `copyStack` and `copylVal` on the parameters to `alloc` and then switched the roles of the two halves of memory.

Congratulations! Once you have completed this last task, you will have a implemented a complete garbage collector. To be sure that you implementation is correct, try running some programs with a heap size that is smaller than the *total* number of large values allocated, but as great as the largest number of reachable values at any time. You can try running `test.mml` with a heap of size 33.

6 Mark-And-Sweep Collection

A Mark-And-Sweep collector intermixes currently allocated cells with free cells on the heap. In order to be able to allocate new cells efficiently, the free cells are maintained as a linked list, that is, the contents of a free cell actually points to the next free cell, etc.

When a Mark-And-Sweep collector starts collecting, it also traces through the stack finding all the reachable locations. However, whenever a location is reached, instead of copying its contents, we mark it as reached (and traverse any children it might have). Once we have marked all the reachable locations in this manner we "sweep" through the whole heap. If a cell was marked, then we clear the marker, if it wasn't marked, then we add it to the free list.

We now need a free list because when we collect we aren't "compacting" all the reachable locations at the beginning of our space (as we were before), so our space can become fragmented.

We have provided you with the following starter code for the Mark-And-Sweep heap:

```
val heap :  HeapElement Array.array ref
val next :  int ref
```

Again `heap` represents our physical memory, but now `next` represents the index of the first cell in the free list, not the beginning of free memory. Our `HeapElement` type has also changed to allow for our new bookkeeping information. `Unmarked` represents a cell which has some data that hasn't been marked by the collection algorithm. `Marked` represents a cell which has some data and *has* been marked by the collection algorithm. And `Free` represents a cell which has not been allocated, and stores the location of the next cell on the free list. For this collector, our `location` type is simply `int`, because we do not move heap cells once allocated.

Task 5: Mark-And-Sweep Allocation (10 points)
Implement the `alloc` function, which should call the collector if `next` is outside of the heap. Otherwise, if should allocate the next available cell (updating `next` accordingly). If the collector doesn't free any memory, then `alloc` should raise the exception `OutOfMemory`. You should also implement the read function, which simply returns the large value associated with the supplied location in the heap.

When we run out of cells in the free list, we must try to "collect" any unreachable memory locations. We do this by tracing through the stack as before. When we reach a location which is unmarked we mark the location and traverse the large value stored there.

Task 6: Tracing/Marking Large Values (10 points)
In the file `mark-gc.sml`, you will find several stubbed out (mutually recursive) functions: `markStack`, `markEnv`, `markPtr`, `marksVal`, and `marklVal`. These functions will work together to traverse the current stack, k, the large value we are trying to allocate space for and mark all the reachable locations.

Task 7: Sweep (10 points)
Implement the function `sweep`, which scans the entire heap doing the following:

- For a `Free` cell, do nothing.
- For a `Marked` cell, change it to `Unmarked`.
- For an `Unmarked` cell, add it to the free list.

Task 8: Collection (10 points)
In Mark-And-Sweep collection, what we will do is mark all of the reachable locations in our heap by tracing through the stack, and the value we are trying to allocate space for. This is taken care of by: `markStack` and `marklVal`. However, after all of the reachable locations have been marked, we must sweep through the heap. This is taken care of by: `sweep`. Implement a function, which you call in `alloc` when you run out of available memory, that calls `markStack` on the supplied stack, and `marklVal` on the value we are trying to allocate space for, and then calls `sweep` to cleanup the heap.

7 Analysis

It is important in choosing what kind of garbage collector to use for your interpreter to perform tests to see how they perform under various circumstances. In order to do this, we will have to set up our collectors to produce certain usage statistics, and we will have to write test cases with which to test our collectors.

Task 9: Usage Statistics (5 points)
Set up your collectors to maintain the following statistics:

- Number of allocations performed
- Number of reads performed
- Number of collections performed
- Number of accesses to heap locations during collection. (*This means we would increment some counter for every read and write to memory which occurs during collection*)

Task 10: Benchmarks (10 points)
Write a function in MinML which implements quicksort on `ints` in the file: `qsort.mml`, and another which implements insertion sort on `ints` in the file: `isort.mml`.

Task 11: Analysis (35 points)
Produce a written analysis of how your garbage collector performs for the two different sorting algorithms with 20 different heap and list sizes. You should include tables or graphs with the performance data produced by your collector. You should also reach a conclusion about the number of accesses required by the two collection algorithms. To aid in your analysis, we have provided a function:

```
Top.file_apply heapSize listSize "test_file.mml";
```

which applies the function given in the file to a random list of the specified length, using the given heap size.

8 Extra Credit

A reference counting collector operates by knowing (for each element on the heap) how many active references there are to it. This is done by incrementing and decrementing the reference counts associated with a large value, as variable which reference it come in and out of scope. Naturally when the reference count reaches zero, we may collect that large value.

Task 12: Reference Counting Collector (30 points)
Program and measure a reference counting garbage collector. For this collector (and this collector only) you will have to change the GC signature so that reference counts can be incremented and decremented by the evaluator.

9 Test Cases

We have included two files, `test.mml` and `test2.mml` that exhibit interesting behavior from a memory usage standpoint, but the results of their respective executions should be obvious.

You are encouraged to submit other test cases to us. We will test each submission against a subset of the submitted test cases, in addition to our own. So, even though you will not receive any points specifically for handing in test cases, it is in your interest to send us tests that your code handles correctly. See below for submission instructions.

10 Hand-in Instructions

Turn in the files `eval.sml`, `semi-gc.sml`, `mark-gc.mml`, `qsort.mml`, and `isort.mml` along with any other test files by copying them to your handin directory

`/afs/andrew/scs/cs/15-312/students/`*Andrew user ID*`/P4/`

by 11:59 pm on the due date. Immediately after the deadline, we will run a script to sweep through all the handin directories and copy your files elsewhere. We will also sweep 24, 48, and 72 hours after the deadline, for anyone using late days on this assignment.

Turn in non-programming questions as a text, postscript or pdf file in the handin directory called `measurement.txt` (or `.ps` or `.pdf`).

For more information on handing in code, refer to

```
http://www.cs.cmu.edu/~fp/courses/312/assignments.html
```