Assignment 6:
Storage Management

15-312: Foundations of Programming Languages
Daniel Spoonhower (spoons@cs.cmu.edu)

Out: Friday, October 24, 2003
Due: Thursday, November 6, 2003 (11:59 pm)

100 points total + 15 points extra credit

Revised October 29, 2003

1 Introduction

In this assignment, you will implement two versions of a garbage collector for an abstract machine. Your garbage collector will automatically manage the memory used to store pairs and closures. Before you begin, you may want to review Harper's chapter on storage management (Chapter 31). 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` and `gc.sml`.

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. You may also have the opportunity to reuse part your solution to this assignment in a future assignment. In each of the latter situations, it is to your benefit to write clean, legible code.

We will be considering exactly the same language we used in Assignment 4, MinML with pairs and exceptions, 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 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:

$$H \mid k \mid \eta > e$$

We might read this judgment as "we are currently evaluating expression e in environment η 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 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. (As in our previous discussions, neither exception values nor locations will be expressed by the programmer in the source language.)

$$\begin{array}{llll} \textit{(small values)} & v & ::= & i \mid \texttt{true} \mid \texttt{false} \mid \texttt{unitel} \mid \texttt{exn}(k) \mid l \\ \textit{(large values)} & V & ::= & \langle\!\langle \eta; e \rangle\!\rangle \mid \texttt{pair}(v_1, v_2) \end{array}$$

Following our comments above, the A machine will also define a *heap*, a finite map from locations to large values.

$$\textit{(heaps)} \quad H \quad ::= \quad \cdot \mid H[l = V]$$

Given a heap, the problem of garbage collection may then be phrased as, "when it is safe to removing 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. Presently, we will be considering one instance of a tracing collection algorithm: semi-space copying collection.

Lastly, but before we begin, you should note that we have made two other changes to the abstract machine with which you became familiar a few weeks ago. We have made the form of each variable binding explicit: each binding is given by an instance of the ML datatype `binding` and is either an ordinary binding (`BOrd`, e.g. $x = v$) or a recursive binding (`BRec`, e.g. $x \stackrel{*}{=} v$). We have also removed the `VSuspend` value, as recursive expressions are now bound explicitly as such.

3 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, functions and recursive expressions.

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 rule for recursive binding lookup has already been implemented for you.) Note that even though we have not begun our implementation the garbage collector, you will still use the functions `alloc` and `defer`, 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 `DBUtils.trim` to ensure that your machine builds the smallest possible closures.

$$H \mid k \triangleright \texttt{pair}(v_1, \square) \mid \eta < v_2 \mapsto_{\texttt{a}} H[l = \texttt{pair}(v_1, v_2)] \mid k \mid \eta < l$$

$$\frac{H = (H_1, l = \texttt{pair}(v_1, v_2), H_2)}{H \mid k \triangleright \texttt{fst}(\square) \mid \eta < l \mapsto_{\texttt{a}} H \mid k \mid \eta < v_1}$$

$$\frac{H = (H_1, l = \texttt{pair}(v_1, v_2), H_2)}{H \mid k \triangleright \texttt{snd}(\square) \mid \eta < l \mapsto_{\texttt{a}} H \mid k \mid \eta < v_2}$$

$$H \mid k \mid \eta > \texttt{fn}(\tau, x.e) \mapsto_{\texttt{a}} H[l = \langle\!\langle \eta; \texttt{fn}(\tau, x.e) \rangle\!\rangle] \mid k \mid \eta < l$$

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

$$H \mid k \mid \eta > \texttt{rec}(\tau, x.e) \mapsto_{\texttt{a}} H[l = \langle\!\langle \eta; \texttt{rec}(\tau, x.e) \rangle\!\rangle] \mid k \triangleright \eta \mid \eta, x \stackrel{*}{=} l > e$$

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.loop_eval` *heapSize*`;` or `Top.file_eval` *heapSize* `"test_file.mml";`

at the SML/NJ prompt. If you set the heap size to a large enough value, you should be able to test your implementation by running any MinML program from either of the previous two programming assignments.

4 Stop-the-World Collection

We will model the behavior of a semi-space collector by tracking two sets of location mappings. In this part of the problem you will use the machine model described the `AMach` structure and modify the garbage collector defined by `StopTheWorldGC`.

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] When that half is consumed, the collector begins to trace through memory, for example from one pair to another. Each time it reaches

[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).

a value for the first time, it copies that value to the reserved half of memory. When all reachable values have been found and copied, we present the (formerly) reserved space to the mutator and continue as before. Any values that were not copied were not reachable and, therefore, could not be part of any future evaluation.

In order to perform this tracing, the collector must be able to discover which parts of pairs, for example, contain locations that point elsewhere in the heap and which are simply integers or booleans. In order to do so, we will define a set of related functions that determine the *free locations* of values, environments, and the stack.

In this case, the collector will only begin work when no free memory remains. At that point in time, it will complete a full collection cycle, reclaiming as much memory as possible before returning control to the mutator. This style of collection is often known as *stop-the-world* collection.

Task 2: Determining Free Locations (5 points)
In the file `gc.sml`, you will find several stubbed out functions; we are currently interested in `FLSV`, `FLE`, `FLLV`, and `FLS`, functions that will determine the free locations of small values, environments, large values, and the stack. Give a definition to these functions and implement them in `gc.sml`. (You need not turn in the definitions separately.)

The G machine

Following the description given in Harper's book you will implement a garbage collector based on a set of transition rules called the G machine. Following our informal description of semi-space collectors above, we will have two sub-heaps H_f, H_t, often known as the *from-* and *to-spaces* respectively, along with a *scan set* S, a set of locations that we are currently processing. The state of the G machine is then given by the tuple

$$(H_f, S, H_t)$$

As you will be implementing a graph traversal algorithm, you can think of the scan set as the frontier of that traversal. The transition rules for the G machine are given below.

$$\frac{}{(H_f[l = V], S \cup \{l\}, H_t) \mapsto_{\mathsf{g}} (H_f, S \cup \mathit{FLLV}(V), H_t[l = V])}\ \mathit{Copy}$$

$$\frac{}{(H_f, S \cup \{l\}, H_t[l = V]) \mapsto_{\mathsf{g}} (H_f, S, H_t[l = V])}\ \mathit{Discard}$$

The first rule *Copy* tells us what to do the first time a location appears in the scan set: we copy the location and its value into the *to-space*, and modify the scan set by replacing l with the free locations of V. If we have already scanned a value (as in the case of a cyclic data structure), then *Discard* says that we can simply remove l from the scan set and continue.

Task 3: G machine Transitions (15 points)
In `StopTheWorldGC`, implement the G machine transitions in the function `step`. This function should make *exactly one* transition each time it is applied. If neither rule above applies and there is at least one free cell in the *to-space*, then collection is complete and `step` should raise the exception `Done`. If neither rules applies and there are no free cells in the *to-space* then the collection was unsuccessful and the collector should raise the exception `Memory`.

Putting them together

Now we simply need to connect our two machine models to get a fully functioning implementation of MinML. The following rules state how the collector may be invoked.[2]

$$\frac{(H, FLS(k) \cup FLE(\eta), \emptyset) \mapsto_g^* (H'', \emptyset, H')}{H \mid k \mid \eta > e \mapsto_{\mathsf{a}} H' \mid k \mid \eta > e} \qquad \frac{(H, FLS(k) \cup FLE(\eta) \cup FLSV(v), \emptyset) \mapsto_g^* (H'', \emptyset, H')}{H \mid k \mid \eta < v \mapsto_{\mathsf{a}} H' \mid k \mid \eta < v}$$

Notice how these rules allow the A machine to invoke the G machine at *any* time. The implementation of the `step'` function in `eval.sml` reflects this: any time the `Collect` exception is raised, we immediately invoke the collector.

Task 4: Invoking Collection (10 points)
Finally, implement the function `collect`, the function that allows transitions from the A machine to the G machine. `collect` should invoke `StopTheWorldGC.multiStep` (which has already been implemented for you). You may use the function `Memory.init` to create a new (empty) *to-space* at the start of collection.

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. For example, the program contained in `space.mml` will run with a heap of size 4, but not with a heap of size 3. Verify this behavior in your collector.

Task 5: Constrained Resources (5 points)
Give a well-formed program in our current version of MinML, but *without* using pairs, that will terminate normally when run with some heap of size n, but that will abort execution with a `Memory` exception when run with a heap of size $n - 1$ (for some reasonably small $n > 1$). Include your program in a file called `constrained.mml` with the rest of your implementation.

5 Incremental Collection

So far we have defined collection in a way that is independent of our abstract machine. That is, the collector's *internal* structure has no effect on the structure of the A machine. While this has some advantages, it means that we must perform an *entire* collection before returning control to the mutator. A more desirable collector might instead carry out its duties one *increment* at a time, interleaving its work with that of the mutator. In this part of the assignment, you will implement exactly such a collector in the structure `IncrementalGC`.

In order to account for the details of our new strategy, we will need to make some changes to our abstract machine. You will now be using the `AGMach` machine representation; we will elaborate in a moment. In the meantime, all you need to do is uncomment two lines in `top.sml`, lines 26 and 27.

Like the A machine, the AG machine will also include a heap as part of its structure, but we must now expose some of the internal details of our collector as well. In addition, we need to account for the fact that sometimes the mutator will be running by itself, while at other times it

[2] As revised in the first correction to the assignment.

will be sharing time with the collector. When the collector is running, states of the AG machine will look something like

$$H_f, S, H_t \mid k \mid \eta > e$$

That is, we must maintain the internal state of the collector explicitly.

In order to simplify the implementation of the evaluator, we will give a single type that describes the state of the collector, regardless of whether it is on or off.

```
type heap = lvalue memory * loc list * lvalue memory * bool
```

That is, the heap will be composed of a *from-space*, a scan set, a *to-space*, and a boolean that indicates whether or not the collector is running. When the collector is off, we will allocate from the *from-space* by convention.

Due to its complexity, we will describe our incremental collector in a more algorithmic nature, but you should note that the core details of the implementation are consistent with the transition rules for the G machine: we are still implementing a semi-space copying collector.

In the course of execution, we would like to maintain the following invariants.

- When the collector is off, allocation will be performed in the *from-space* exactly as before but with the following exception: we may begin collection even if there is enough free space to satisfy the current request.
- When the collector is off, we check to see if we should be begin collection by inspecting the *from-space* H_f during allocation. If the *from-space* is more than half full, then we signal a new collection by raising the `Collect` exception.
- When the collector is on, we perform allocation in *both* semi-spaces with the *same* location. This ensures that the new large value lives through to the end of the current collection and that we are not re-using a memory cell in the *to-space* that will be required for some other value (yet to be copied).
- When the collector is invoked for the first time (after running in the "off" mode) it should initialize the scan set, just as in our first collector. In this case it does *not* need to perform any further work (presumably the initialization itself forms an "increment").
- Otherwise, when the collector is invoked, it should do the work corresponding to one step of the G machine with the following exception.[3]

$$\frac{l \notin \mathrm{dom}(H_t)}{(H_f[l = V], S \cup \{l\}, H_t) \mapsto_{\mathtt{g}} (H_f[l = V], S \cup \mathit{FLLV}(V), H_t[l = V])}\ \mathit{Copy}'$$

Task 6: Allocation (10 points)
Implement allocation in the function `alloc` in `IncrementalGC` according the invariants above.

Task 7: Single-step Collection (25 points)
Implement a single step of garbage collection in the function `step` maintaining the invariants

[3] As revised in the second correction to the assignment.

above. **[Hint: Part of your implementation will look very similar to your original implementation of the G machine transition rules.]**

Task 8: Invoking Incremental Collection (10 points)
Just as before, tie the knot by completing the implementation of the `collect` function, again, maintaining the invariants above. Note that there is no analog of `multiStep` in an incremental collector, so you should invoke `step` directly.

At this point, you've completed a second garbage collector, though (as is common in many incremental collectors) your implementation may require larger heaps for a given program than the stop-the-world collector above. This is part of the trade-off of incremental collectors, we may pay a higher price for garbage collection, but we pay it in smaller increments.

Additional invariants

One of the more difficult aspects of implementing an incremental collector is ensuring that the two semi-spaces are maintained a consistent manner. This is particularly problematic in language where the mutator can actually *update* values in the heap. One strategy for dealing with this problem is to force the mutator to operate only on values in the *to-space*.[4] Then, if the mutator updates any data structures during collection, we know that the version in the *to-space* reflects all of these changes.

Even though our current version of MinML is a pure language, we will make the appropriate changes to implement this invariant. In order to do so, we will implement a *read-barrier*. That is, when the collector is on, we will perform a check at each dereference operation ensuring that a copy of the desire value lives in the *to-space* (though an identical copy may also reside in the *from-space*).

Task 9: Read Barrier (10 points)
Change the implementation (but not the type!) of `deref` in `IncrementalGC` to enforce that, if the collector is on, any value returned to the mutator lives in the *to-space*.

6 Optimization

One form of optimization you should already be familiar with is *tail-call elimination*. While we will not be able to eliminate tail-calls as we might in an imperative language, the explicit stack of the E and A machines will allow us to improve the performance of our interpreter by recognizing these calls and treating them in a manner different than ordinary function application.

Task 10: Tail-Call Optimization (EXTRA CREDIT, 15 points)
For full credit, complete each of the following subtasks:

- Give a program that includes at least one function application in tail-call position and one that is not. For both cases, give an (abbreviated) description of the machine state just before the application is carried out (i.e. the top stack frame should be of the form `apply(`$v_1, \square$`)`). You only need to write the relevant parts of the heap, stack, and environment.

[4] This is known a *to-space invariant*.

- Modify or extend the transition rules for the A machine to recognize tail-calls and perform the optimization eluded to above. Hand in your rules electronically as `tail-call.txt` or `tail-call.ps` or submit them in written form. Make a duplicate `tail-call-eval.sml` of `eval.sml` and implement your rules in the copy.
- Explain how this optimization will improve the performance of the garbage collector. Give an example of a well-formed MinML program that will terminate normally with the optimized transition rules, but will raise a `Memory` exception with the original rules (with the same heap size in both cases). Include your program (and the size of an appropriate heap) in a file `tail-call.mml`.

7 Test Cases

Since we have not changed the definition of our language, any test cases from previous assignments should still have the same behavior. We have included two files, `space.mml` and `copies.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 (beyond those points received in Task 5), it is in your interest to send us tests that your code handles correctly. See below for submission instructions.

8 Hand-in Instructions

Turn in the files `eval.sml`, `gc.sml`, and `constrained.mml` along with any other extra credit and test files by copying them to your handin directory

`/afs/andrew/scs/cs/15-312/students/`*Andrew user ID*`/asst6/`

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 text or postscript files in the handin directory. Or, if you wish, you may turn in answers on paper, due in WeH 5119 by 11:59 pm on the due date. If you are handing the assignment in late, either submit your solution electronically or bring it to 5119 Wean. If I am not in my office, write the date and time at the top and leave it on my chair or (if the door is closed) slide it underneath the door.

Also, please turn in any test cases you'd like us to use by copying them to your handin directory. To ensure that we notice the files, make sure they have the suffix `.mml`. You should include a comment in the file that indicates how large the heap should be in order for the program to run correctly.

For more information on handing in code, refer to

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