Extra Credit*

CS4536    CS536

Due Noon, Thursday 1 Mar.

Goal of this assignment. This assignment offers four separate extra credit activities. Each activity is worth a maximum of 50 points, i.e. half a regular homework project. You can work on any set of these activities, handing them in to receive up to the sum of the points for the problems you’ve chosen. Two concern garbage collection. One concerns typing. The last one concerns a different way to implement proper tail recursion. All four are here now (26 February, 8 pm).

Correction. I’ve now (28 February, 11 pm) corrected the description of mark-and-sweep to emphasize that in the forward and backward phases, there are two parameters. One will always indicate the current heap object. The other indicates its predecessor, namely the heap object that we reached this one from.

Note to CS 536 students. Since CS 536 continues, and I do not have to give you a grade until the end of Spring semester, you can receive a maximum of 40 points for each of the problems, if handed in by Midnight, end of 12 March.

1 A new twist on mark and sweep GC

There is a practical problem with the basic mark-and-sweep algorithm: It uses depth-first-search (DFS) to mark all the accessible nodes. Unfortunately, DFS requires a stack, either as a data structure manipulated in the program itself, or (usually) via the control stack of the language used to implement DFS. The stack remembers which node we should return to, when

---

*This is version 3, containing all four problems, and a correction to the mark and sweep description. 28 February, 11 pm.
we’re finished exploring the graph from the current node. This stack grows as DFS makes recursive calls. Its height at its maximum will be similar to the length of the longest pointer-to-pointer chain in the heap. But that can be very long, possibly most of the heap. Thus, we would need a lot of spare space to do our garbage collection.

Not that long after McCarthy invented garbage collection, Schorr and Waite published a clever trick to avoid using all this spare space (CACM, 1967; also attributed to Deutsch). The idea was that instead of using a separate stack to remember what node to return to after finishing a given node, we will store a back pointer to the place we came from right in the node itself.

Where will we get the space from in this node? If the depth first search is going to continue, the current node has a pointer onward to a new node that we have not yet marked. So we can remember that pointer, and insert the back pointer into its slot, and make a tail recursive call to go forward. In the forward call, we need two parameters, one being the node that becomes our new current node, and the other being the previous node that we have just arrived from.

We will also make the mark into an integer, whose value will keep track of how many fields in an object we’ve already considered. For instance, if an object is a vector of length \( k \), then the mark is 0 when we still need to start with entry 0 in the vector; when the mark is \( k \), there are no more entries to look at, and the mark phase is done with this vector.

This means that the DFS requires two phases.

**Forward** In the forward phase, we follow a pointer from source to target, remembering the source as an extra parameter. We do this only if the new target is not yet marked.

Before we leave the source, we store a back pointer to its predecessor inside the source. Since we’re doing garbage collection, we flip the marked bit inside target’s header to true.

Since we’re proceeding depth-first, we now follow a pointer from target (if it contains one), in another forward step, where this node will be remembered as the predecessor.

**Backward** If we don’t have any pointers to follow forward, then we take a backward step. In the backward step, we look up the back pointer in the predecessor node. We restore the pointer to the current node where the back pointer was. And then we call the *forward* step with
type heap_entry =
  (* This word of heap is a header *)
  Hdr of header * gc_data
  (* This word is not a header, but an obj *)
  | Obj of pl_object
  (* Back pointer used in mark_and_sweep *)
  | Back of int

Figure 1: Expanded definition of heap objects

the predecessor node as the new current, and its predecessor, which we recovered from its back pointer, as the new predecessor.

If the target of the back pointer is a location mid-way through its object, then we look to see whether there are any more pointers to explore from in that object. If so, we take a forward step from there; otherwise, a backward step from there.

In this process, we must be careful that all the procedure calls to forward and backward are tail-recursive, so that we are not accumulating data on the stack.

When you implement this, you will want to modify vm_data.ml. You will want to add a new kind of heap entry, namely the back pointers, so that they are easy to distinguish from regular pointers (heap objects). When we are not in the mark phase, the heap contains no back pointers. They only exist while we are in the midst of the mark phase. Your new heap entries may be defined as in Fig. 1. When you work your way back to a heap object that has no back pointer, then the DFS for a particular garbage collection root is complete.

Implement this improved version of mark-and-sweep, and test it on several heaps.

2 Stop and Copy GC

In copying GC, we use two different heaps. At any time, one of them is active, and the other is not in use. During GC, we copy all of the active data from the previously active space (which we will call from-space) to the inactive space (which we will call to-space).

We use breadth-first-search as the underlying traversal strategy, effectively storing the BFS queue inside to-space itself. The end-of-queue pointer
is identical with the free-space pointer, which points to the first word of to-space that we have not yet used. The start-of-queue pointer points to the beginning of the first object that we have copied into to-space, but which may contain pointers back to from-space, pointing to objects that we have not yet copied.

The insight that copying garbage collection can use BFS with the queue in to-space is due to C. J. Cheney (CACM, 1970).

Starting at each of the roots, i.e. pointers from the control state into the heap, we take the following steps:

1. Given a pointer \( p \) into from-space, we look it up in from-space. There will be a header at \( p - 1 \), which gives the length \( \ell \) of the object.

   We copy from \( p - 1 \) through \( p - 1 + \ell \), i.e. the whole object including its header, to the end-of-queue pointer \( q_{\text{end}} \).

   When we copy the object \( p \) points to, any pointers within it become pointers into from-space.

   In the first word of the object that \( p \) points to in from-space, we place a relocation pointer to \( q_{\text{end}} + 1 \), immediately after the header in to-space that we have relocated object \( p \) to. This relocation pointer is often called a broken heart\(^1\).

2. We update \( p \) to point to \( q_{\text{end}} + 1 \), and then we increment \( q_{\text{end}} \) by \( \ell + 1 \) to point past the end of the newly copied object.

3. We now look at the start-of-queue pointer \( q_{\text{start}} \). If the value at \( q_{\text{start}} = q_{\text{end}} \), then we are done copying.

   If \( q_{\text{start}} \) is an immediate, non-pointer value, or a header, then we increment \( q_{\text{start}} \).

   If \( q_{\text{start}} \) is a pointer into from-space, and it points to a broken heart, then we update it to the object in to-space that the broken heart leads to. We now increment \( q_{\text{start}} \).

   If \( q_{\text{start}} \) is a pointer into from-space, but does not point to a broken heart, then it points to an object we must copy into to-space. To do so, go to step 1 (incrementing \( q_{\text{start}} \)).

Implement this and test it on three heaps, including the two in some_heaps.ml. You will want to extend pl_objects in vm_data.ml to include a new kind

---

\(^1\) Presumably because other pointers to this now-moving object have lost the object of their affection; the broken heart pointer tells them where to look to find the beloved again.
type pl_object =
  True |
  False |
  Dummy |
  Int of int |
  Heap_obj of int |
  In_channel of in_channel |
  Out_channel of out_channel |
  Cross_heap of int (* pointer into other heap *)

type vm_state =
{ code : int array;
  arg_stack : pl_object Stacks.t;
  ctl_stack : int Stacks.t;
  heap : mutable heap_entry array; (* use this one now *)
  spare_heap : mutable heap_entry array; (* keep this for GC *)
  mutable free_ptr : int;
  mutable loc_of_empty_env : int }

Figure 2: Types for stop-and-copy

let v = ref 5 in
let w = ref 7 in
let z = ref w in
(get v) + ((get z) := ((get (get z)) + 1))

Figure 3: Simple cell 3

of pointer, namely a cross-heap pointer Cross_heap of int. You will also enrich the type vm_state to contain a second heap.

3 Extending typing to handle let-polymorphism

In project 3, not all of the examples type-checked. In a couple of cases, this was because the code used primitives that I didn’t ask you to implement, specifically the vector primitives. They would be easy to add.

But there was also another problem, in simple_cell3.txt, that requires more work (see Fig. 3). Here, the ref operator is applied to integers, the produce cells that hold integers. It is also applied to a cell, namely the cell
w, that holds an integer, to produce a cell that holds a cell that holds an integer.

The way that typings.ml uses unification, this can’t work, because ref has some type Ref $\alpha$, and once we have unified $\alpha$ with int, we cannot also unify $\alpha$ with Ref int.

Milner’s solution to this was to allow operators to be freshly instantiated at several different types. He formalized this with a type scheme which is essentially a type in which some type variables are “universally bound.” For instance, if $\tau$ is a type, then $\forall \alpha . \forall \beta . \tau$ is a type scheme. We can get a fresh copy of this whenever we want, in which $\alpha$ and $\beta$ are instantiated with new type variables that do not appear in $\tau$, or anything we are working with. Thus, unification can re-start anew with them.

Implement this version of Milner-style polymorphism, and use it to correctly type Fig. 3 and a new example that you create. The code in type_schemes.ml in the distribution for Project 3 will be useful, since it gives type definitions, and some function definitions for getting a fresh instance of a type scheme.

4 Optimizing tail recursion after compilation

In our treatment of tail recursion, we identified the opportunity to do a tail call—rather than a call that pushes a closure on the stack and requires an additional return—in the main compiler algorithm.

An alternative approach is also possible: We could do the main compilation without worrying about identifying tail recursion. Then, when it is done, we could inspect the resulting byte-level abstract syntax tree. If we can see that a blast does a call, and then immediately afterwards does a return, we can replace that call with a tail call.

Of course, sometimes the blast may do a call; when the callee returns, the blast may branch before doing its return. In this case also, we may as well replace the call with a tail call, thereby avoiding one return and also a branch. The essential effect is the same.

The goal of this activity is to implement tail call as a blast-to-blast transformation after compilation is finished. The compilation can therefore be blind to tail vs. non-tail calls.

Working with the Lua code in min_tr/lcompile.lua:

1. First determine what actions are like branching. That is, what blast instructions, situated between a call and the return point in the blast, do nothing of value? Or more precisely, which ones have no effect that
will still be visible after the current blast returns? Let us call these the *terminally ignorable* instructions.

2. Next write a procedure to act on a blast, and for each location where there is a call, determine whether every instruction between the call and a return terminating the blast is terminally ignorable.

3. Then, write code to generate a blast which contains, in place of each of these calls, a tail call.

4. Write up a brief explanation of why: If the blast code input to your transformation computes a given result, then the code you output must compute the same output.

   Would you be able to show that (a) your transformation identifies all the tail calls that the tail-call-aware compiler identifies? (b) that your transformation correctly identifies more tail calls than the tail-call-aware compiler?