Compiler Construction - Supervision 4

Nandor Licker

October 2019

1 Past Papers

- 2018 Paper 4 Question 3
- 2018 Paper 4 Question 4
- 2017 Paper 4 Question 3
- 2017 Paper 4 Question 4

2 Practical

2.1 Garbage Collection

You might have noticed that the language and its implementation is not particularly memory efficient: this task should fix that. By now, two methods should be allocating memory in the runtime: the one allocating closures and the one allocating objects. To implement the garbage collector, these methods will be modified to first check if they can perform the allocation: if not enough memory is available to satisfy the current request, garbage collection is triggered to attempt to free up space.

The simplest garbage collector, which will be described here, is a two-space copying collector based on Cheney's algorithm. The method operates in two steps: first, the heap is traversed starting from the roots on the stack, marking all objects as visited. Then, the live objects are copied from one half of the heap to the other, rewriting all addresses. The following steps should lead you to a functional collector:

- Start by adding a C file to the runtime: implementing a garbage collector in assembly is not particularly fun and this is not 1959. Modify the `lambda` script to link the C runtime into the output.
- Define two variables in the C file: `gc_stack_top` and `gc_stack_bottom`. These two will delimited the stack region managed by the GC, allowing all roots to be identified.
- Define another pair of variables, `heap_current_ptr` and `heap_limit_ptr`. Allocators in garbage collected languages are usually bump pointer allocators: they increment a pointer by the number of bytes they allocate, triggering GC when the current pointer hits the limit.
- In the main method, initialise `gc_stack_top` with the address of the stack pointer: you might need to offset it a bit.
- Define a `gc_init` pointer and call it from the main method. This will initialise the GC.
- Define a `gc_collect` method which will be called from the assembly method.
- In the methods which allocate, determine the allocation size and check if it fits between the current pointer and the heap limit pointer. If it does not, trigger GC first. After the GC call, return the previous current pointer as the address of the allocated block: offset it by the size of the header. Increment the current pointer by the appropriate size.

- The marking phase requires a few bits in object headers: define a layout which can accommodate the mark bit, the tag and the block size. Adapt existing code to work with this layout.
- It is necessary to distinguish integers from objects. As an example, OCaml uses the last bit of a 64-bit value: 1 indicates primitives, 0 indicates pointers. Adjust all integer constants and methods operating on integers to work on this representation. For example, the constant opcode should encode $n * 2 + 1$ instead of n. The `Add` instruction should now be implemented as:

  ```
  popq %rcx
  addq %rcx, (%rsp)
  decq (%rsp)
  ```

  Since a and b are represented as $a * 2 + 1$ and $b * 2 + 1$, $a + b$ should be represented as $(a + b) * 2 + 1$.
- Enumerate all pointers from the stack: traverse all 64-bit values between the top and the bottom of the stack, as defined by `gc_stack_top` and `gc_stack_bottom`, excluding primitives, base pointers and return addresses.
- Implement the mark phase, setting the appropriate mark bits. You might choose a simple depth-first traversal or opt for a more complex implementation which does not require additional stack space. During the mark phase, you might encounter pointers which are outside the GC-managed heap, such as function pointers and closures for functions defined in the language. Do not enter them.
- Implement Cheney's algorithm: forward all root pointers from the stack and all live objects to their new addresses in the compacted heap.