HOT-Compilation: Garbage Collection

TA: Akiva Leffert
aleffert@andrew.cmu.edu

Out: Saturday, December 9th
In: Tuesday, December 19th (Before midnight)

1 Introduction

It's time to take a step back and congratulate yourself. You've implemented large parts of an SML compiler. The stages after closure conversion — hoisting, allocation, code generation, are not very interesting, at least with a naive implementation. However, even after code generation, the job of the compiler writer is not yet done. Modern languages typically provide automatic memory management facilities packaged as part of a runtime. This constitutes an allocator and a garbage collector. In this assignment you will be implementing a simple garbage collector. More specifically, you will be implementing a copying garbage collector using the Cheney scan algorithm.

2 Overview

For this assignment, you'll need to dust off your C hacking skills. You will implement the functions declared in `runtime/gc.h`. Prototypes for the two functions are reproduced here:

```
hotcint_t gc_init(hotcint_t space_size,
                  hotc_t** allocp,
                  hotc_t** limitp);
void gc_collect(hotc_t* root,
                hotc_t** allocp,
                hotc_t** limitp);
```

Implementation of these functions should go in `runtime/gc.c`. We will explain the purpose of and parameters to each of these functions in Section 5. You may find useful macros defined in `runtime/runtime.h`.

We provide you with the skeleton of a compiler based around the code you have written for the previous assignments and will write for this assignment. Simply drop your assignment solutions into the appropriate directory. Once you have run `make`, you can invoke it with the following command:

```
./hotc <filename>
```

This command accepts the flag `-o <outfile>` to control the name of the output binary. The default output binary name is `a.out`.

Despite the presence of this compiler, we recommend you test your code using small C programs as they will be easier to understand and trace. To build a C file named, for example, `test.c`, run the following at the command line:

```
gcc test.c -Iruntime runtime/runtime.c runtime/gc.c -falign-functions
```

Executables generated by this take an optional integer argument which represents the size of the heap in bytes. The default value is 10000. We include a selection of tests in the `tests` directory. Do not consider your implementation exhaustively tested if you pass those tests. Additionally, You many find the `-g` flag to `gcc` useful. It generates debugging information for use with `gdb`. Also, note that the runtime only works on 32-bit machines and is only guaranteed to work on the andrew system.

Submit your code via AFS by copying `gc.c` to the directory

```
/afs/andrew/course/15/501-819/submit/<your andrew id>/gc/
```

Note: Your submission will be graded automatically; if you submit a `gc.c` that fails to compile under the version of GCC on andrew (3.4.6) using this assignment's base distribution, we won't be able to grade it.

3 Memory Management

In this section we explain how an allocator based around a garbage collector works. We then explain the structure of a copying collector. While you will only be implementing a collector, the collector will need to allocate memory during the copying phase of a collection as well as manage state used by the allocator so you'll need to understand how it works.

3.1 Allocation

We use a linear allocation scheme. That is, the allocator assumes that it has a contiguous region of memory and simply steps through that region as it recieves allocation requests. It keeps a pointer to the beginning of the unassigned part of the region. This pointer is the *allocation pointer*. When the allocator receives an allocation request it simply increments the allocation pointer by the size of the request.

We make the (reasonable) assumption that memory is not infinite. As such, the allocator can reach the end of its memory region. The allocator further assumes that it has a pointer to the end of the allocation memory region so that it can determine when it has reached the end. This is the *limit pointer*. If an allocation would cause the *allocation pointer* to move beyond the *limit pointer*, the allocator will invoke the garbage collector, hopefully clearing enough space to make the allocation. The collector will return a new *allocation pointer* and *limit pointer* when invoked. If, after a collection, there is still not enough space, the program exits with an error. For our implementation of an allocator, and the one which you'll be testing against, see the `hotc_check` and `hotc_alloc` functions in `runtime/runtime.c`.We decouple the checking and allocation stages so that all of the checking can be done at the beginning of a function, combining multiple checks into one big check with the cost of just one.

3.2 Copying Collection

Typical semispace collectors, including the one you will implement, divide memory into two regions called the *from-space* and the *to-space*. Such collectors are called semispace collectors. The two blocks of memory often exchange roles during the run of the program. Allocation is always done in the current *to-space*. Thus only half of memory can actually be used at any one time.

During a collection, the *from-space* and the *to-space* swap pointers. Then all reachable objects are copied from the new *from-space* to the new *to-space*. A reachable object is one which may be used by the program at any point in the future. The collector assumes there is a *root pointer* which is connected to all reachable pieces of memory by traversing the memory graph starting at the root.

The obvious way to copy all of the reachable objects is to perform a depth first traversal, starting at the root node and copying each thing it points to and so on recursively. The things an object points to are known as its children. This strategy has three problems. First, such a traversal can loop forever if there is a cycle in the memory graph. Second, if two objects point to the same object, a naive traversal would make two copies of that object. Third, such a traversal uses memory linear in the depth of the traversal. Putting this memory on the stack is a problem because a deep memory graph would cause a stack overflow.

Putting it on the heap would require having another memory manager just to do memory management, which is possible (and indeed your code will need to call `malloc` during initialization), but undesiriable for code, runtime, and memory complexity reasons. The Cheney scan algorithm solves all these problems.

Before we state the algorithm in pseudo-code, we will introduce the basic concepts. The first is that of a forwarding pointer. This will be used to solve the problem of multiple copies and memory cycles. The idea is to have a data structure that indicates if an object has been copied and if so, to where. The new location of that object is called the object's forwarding pointer. With a forwarding pointer structure, before copying an object, we can see if it already has a forwarding pointer. If it does, instead of copying, we simply overwrite the pointer to the object with its new location. If the object does not have a forwarding pointer then we copy it and set its forwarding pointer before traversing the object's children. In this way we never copy an object more than once and object aliases remain synchronized. It happens that an object's forwarding pointer typically doesn't need to be put in an auxiliary data structure but rather can be written over the header of the old object since its data has been copied somewhere else. You should take advantage of this in your collector.

The Cheney scan algorithm replaces the linear memory usage of a typical DFS with a single extra pointer. This pointer is called the *scan pointer*, or sometimes the *gray pointer*. After we copy an object using this algorithm we do not immediately copy the object's children as we would in a DFS. Instead, we use a different method of traversal. All objects we have copied so far exist in the *to-space*. Therefore we can simply linearly step through the objects in the *to-space* starting at the beginning. Think of this as the queue in a breadth-first search. Each copied object gets put in the end of the queue (the allocation point in the *to-space*), and we trace its children only when we reach it in the queue with the scan pointer. After we copy an object (or not if it has a forwarding pointer) we simply increment the *scan pointer* to the start of the next object. When the scan pointer reaches the allocation pointer we know that all the objects in the *from-space* have been copied.

Pseudo-code for the garbage collection algorithm appears in Figure 1. Your actual code should look similar to this but handle several important special cases. Relevant implementation details will be explained in the next section.

4 Value Representations

The HOT Compiler converts all values into one of two forms: pointers to tuples and everything else. We will call the second type of value a primitive though there is nothing necessarily primitive about them. Such a value may be hiding a large complicated data structure, just one the garbage collector shouldn't try to traverse through. All these values are the size of a word - 4 bytes on a 32-bit architecture. Only pointers to tuples will have children (the elements of the tuple). Thus, you will need to modify the pseudocode in Figure 1, which assumed all objects were of the same form. Pointers to tuples have the lowest bit set to 0. Primitives have the lowest bit set to 1. This means, among other things, that the default for arithmetic is 31-bit with the lowest bit as 1 and the integer value shited left 1 bit. Primitives should not be recursed through by the garbage collector. For example, your collector should recognize a 31-bit integer like 10, represented as the tagged, 32-bit value 21 ($((10 << 1) \mid 1 = 21)$), by seeing that the lowest bit is set and know not to copy it when the `copy` part of the algorithm is called on it, but rather pass the value back out unchanged.

We assign all values the type `hotc_t` which is declared in `runtime/runtime.h` and is duplicated in Figure 2. This type is declared as a union of many things to lessen the burden of casting within the runtime - don't be distracted by the number and name of fields. All representation information you need to know about will be described in this document. To access subfields of an object you can use the different selectors. For example, assume for the moment that the size of a pointer to a tuple is stored in its first element (the actual details are slightly more complicated than that and will be explained in the next section. To access this value for some pointer `p` you would say `p.arr[0].num`.

```
collect:
  to_space, from_space = from_space, to_space
  alloc_pointer = to_space
  limit_pointer = alloc_pointer + space_size
  scan_pointer = alloc_pointer
  *root = copy(*root)
  while scan_pointer < alloc_pointer:
    for p in children(scan_pointer):
      *p = copy(*p)
    scan_pointer = scan_pointer + size(scan_pointer)

copy(p):
  if is_forwarded(p):
    return get_forwarding_pointer(p)
  else:
    memcpy(p, alloc_pointer)
    new_address = alloc_pointer
    forwarding_address = alloc_pointer
    set_forwarding_address(p, alloc_pointer)
    alloc_pointer = alloc_pointer + size(p)
    return forwarding_address
```

Figure 1: Cheney scan copying collection algorithm

```
union hotc_t {
    hotcint_t num;
    hotcstr_t str;
    hotcchr_t chr;
    union hotc_t* arr;
    void* fun;
};
```

Figure 2: Internal universal type

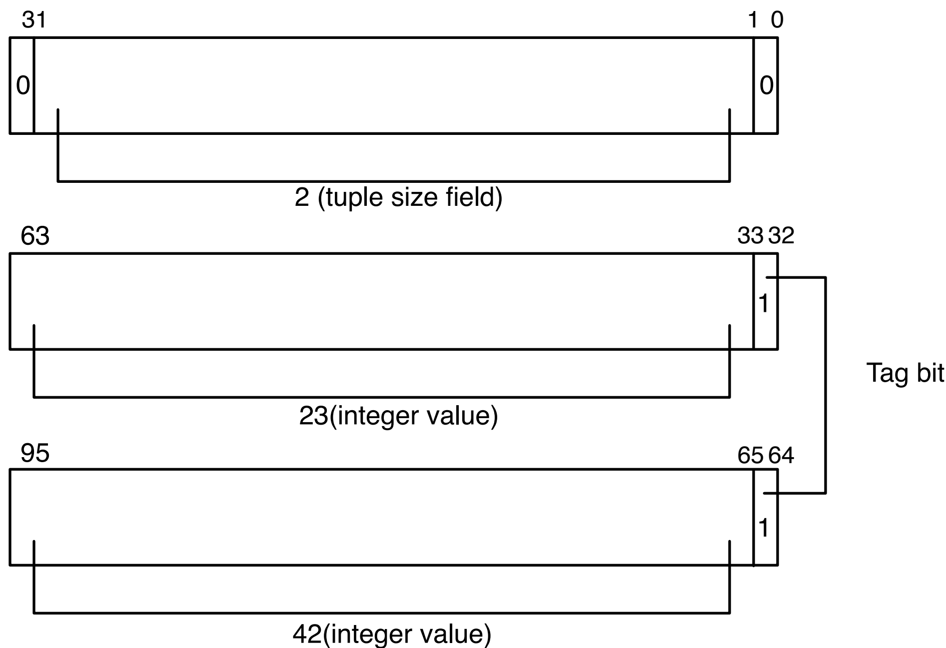

Figure 3: Layout of the tuple (23, 42)

4.1 Tuple Layout

Tuples are represented as a C array where the first element is a header field followed by a sequence of words of the indicated size. The header field keeps track of the number of words which follow the header field. Thus, the total size of the array in words is that size plus one. However, the header field is not just an integer, its lowest bit and highest bit are significant. Thus the actual size value is stored in the middle 30 bits as an unsigned integer.

The lowest bit is reserved for your use. We suggest you use it to mark whether you have overwritten the header field with a forwarding pointer. This usage will never conflict with the bits of a pointer as the allocator always returns word aligned pointers. The high bit is an indication for the garbage collector not to trace the children of the object. If this bit is on, the collector should not recurse through the children of an object when stepping through the *to-space*. We call such objects *untraced*. This allows the runtime system to create objects that it does not want the garbage collector to trace through like any other value. This protection enables the implementation of values where the lowest bit is not 1, allowing the implementation of things like 32-bit integers and strings. Figure 3, presents the memory layout of the tuple $(23, 42)$. When examining this diagram, remember that integers are tagged, so that they are shifted up 1 bit, with the lowest bit set to 1. Figure 4 shows the null-terminated string "hello" as an untraced tuple. Since the text of the string is in an untraced tuple, the collector shouldn't try to trace every four characters as if they were a value or pointer.

5 Implementation

As mentioned in the overview, you will implement two functions, plus whatever helper functions and global data you may need.

Now that you understand the details of copying collection, we can explain the interface properly.

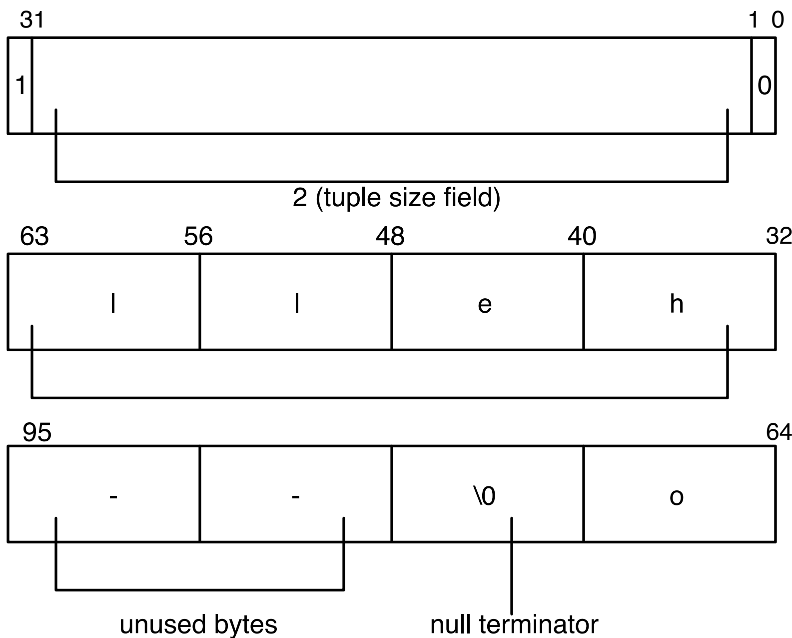

Figure 4: Layout of the tuple "hello"

- ```
  hotcint_t gc_init(hotcint_t space_size,
                    hotc_t** allocp,
                    hotc_t** limitp);
  ```

  The first argument to `gc_init` is the size, *in bytes*, of each semispace. You will need to allocate two regions, the starting *from-space* and the starting *to-space*. Each of these should be of this size. You should do this using `malloc`.

  You should place the start of the *to-space* into the `allocp` argument and the end of the *to-space* into the `limitp` argument. Signify any initialization errors, such as malloc failures by returning a non-zero value. A return value of zero signifies successful initialization of the garbage collector.

- ```
  void gc_collect(hotc_t* root,
                  hotc_t** allocp,
                  hotc_t** limitp);
  ```

The `gc_collect` function is the function which should implement the collection algorithm previously described. The first argument is a pointer to the *root pointer*. The second and third arguments should receive the new *allocation pointer* and *limit pointer* respectively. After a call to this function, the *to-space* should contain no garbage and be ready for allocations starting at *allocp*.