

CS 577 Homework 7 – Copying Garbage Collection - due 10am, Tuesday, June 3, 2008

The goal of this assignment is to write a simple copying garbage collector for a Java interpreter, following the pseudo-code given in lecture. You are provided with a working interpreter, similar to (the solution to) homework 1, but extended to support creation of simple objects (no inheritance) with fields that can be read or written. In addition, the interpreter maintains (though it currently does not use) type information about each stack slot and local variable. The interpreter currently allocates objects and arrays from a simple heap; your job is to add a two-space copying garbage collector, and to test the resulting system thoroughly. For extra credit, you may choose to implement an incremental (“Baker-style”) collector.

Details

In order to have some more interesting heap data structures to collect, the provided interpreter (`interp.c`) has added support for opcodes `NEW` (to create objects), `GETFIELD`, `PUTFIELD`, and `INVOKESPECIAL` (to call constructors). Programs may now include multiple classes, whose class files will be loaded automatically from the current working directory. However, there is still no support for dynamic methods; only static methods (which must still be in the initial class specified on the command line) and constructors can be invoked. Inheritance is not supported; more precisely, all classes must inherit directly from the `Object` class (which is what they do implicitly when no `extends` keyword is used). Support for calculating sizes, offsets, and type information about objects is in auxiliary file `objects.[ch]`.

Heap management and object allocation and access are implemented in `runtime.[ch]`. In particular, note that all operations that read or write arrays or objects are now done by functions defined in `runtime`. As distributed, the system does no garbage collection; it simply allocates objects and arrays from a simple heap array. However, the code has been organized to make it easy to alter heap representations and slot in a garbage collector just by changing the implementations in `runtime.c`, without altering their interfaces or `interp.c`. In particular, `initialize_heap` passes essential information about the *root set* that is currently ignored but can be used by a collector. The first parameter of `initialize_heap` is the total heap size; the user can specify this by an optional second argument on the `interp` command line. The second and third parameters specify the first and last entries in the root set. For this interpreter, the root set is simply the value `stack` (recall that this contains both operand stack slots and local variables). Since the size of the stack changes dynamically, we pass the *address* of the stack pointer, so that the collector can read the current value when needed.

In order to support precise collection, it is also necessary to provide the collector with type information so that it can distinguish pointers from non-pointers. To do this, the interpreter has been modified to maintain a dynamic `typestack` in parallel with the value `stack`; the address of the `typestack` is passed as the fourth argument to `init_heap`. Tracking type information in this way is simple, though obviously not very efficient. (Since we know that verified bytecode assigns a unique type to each stack slot and local at each program point, a more realistic system could do an initial pass over the bytecode to compute and save this type information once and for all — possibly as part of the verification process.) The collector also requires type information for the fields of objects; this can be obtained using the function `object_slot_types` in `object.[ch]`.

Types are represented by their standard class file descriptor strings (see the JVM specification

for details). In fact, the first character of the descriptor string typically suffices to tell the collector whether or not it is represented by a pointer, but we maintain the whole string for generality; it is up to the `runtime` implementation to decide how to interpret it. Note that although we are collecting type information dynamically, it is still necessarily somewhat imprecise because of the possibility of subtyping (even though we only allow trivial inheritance from `Object`). For example, the `ACONST_NULL` operation always pushes type `java/lang/Object` on the type stack; in fact, the `null` may represent any object or array type. Fortunately, raw integers are *not* a subtype of `Object`, so no confusion about pointers vs. non-pointers should result.

Note also that any object type may have the value `null`. This is fortunate, because object storage is created (using `NEW`) well before it is initialized (using a constructor invoked by `INVOKESPECIAL`), and a collection may happen in the interim. As long as `alloc_object` takes care to initialize all object fields to `null`, the collection will be safe.

What to Do

To implement a copying collector as outlined in lecture, you'll need to make the following modifications to `runtime.c`:

- Modify the record layout for arrays and objects to support collection; typically, this means adding a header field to both kinds of records, and perhaps a length field to objects. Decide how to represent forwarding pointers. Make sure that any added fields are correctly initialized!
- Change the heap initialization to create two semispaces instead of one large one.
- Change the `heapalloc` function to do a collection if no more room is available in the current space.
- Implement the copying collector itself.

The collector does not require a lot of code, but experience shows that it can be difficult to get this code right! So you should also build (optional) debugging printout into your program. At the least, it should be possible to report when collections occur and how much live data is found at each. You may also find it useful to provide a function to print out the entire contents of the heap on demand. A heap “sanity checker” that makes sure all pointers are into the correct semispace is also helpful. You can control these features using simple `#IFDEF`'s, or even just comment them in/out. Please don't change the main program to add debug control flags, etc.

For extra credit, you may wish to attempt a Baker-style incremental collector. If you want to pursue this, you might start by reading Baker's original paper (see the course web page).

You must also create and submit a small suite of test programs that exercise the collector. Your tests should cover all the collector code; for example, be sure to exercise forwarding pointers. Hint: keep your heap size small when testing!

How to submit your homework

Submit the homework *by email* to `apt@cs.pdx.edu` with subj line “CS577 HW7” prior to the beginning of class on the due date. You should submit a revised version of file `runtime.c` and your test `.java` files, as *attachments* to your mail.