CS4201
Programming Language Design and Implementation

Garbage Collection

ID:150013828

November 29, 2017

1 Overview

In this report an outline of an implementation of Henry G. Baker's *treadmill* garbage collector is provided. Design decisions and implementation details are then presented for: the required *Episcopal* objects; the underlying allocation model and API; and finally the treadmill collector itself. Finally, the garbage collector is analysed for time complexity in section 5.

2 Building for Testing & Profiling

The solution is built using *gradle*, which will download all necessary dependencies, compile, and run automatically. A gradle wrapper "gradlew" is included with the solution.

In order to run the unit tests provided (see the directory "src/test/java/", also see section 4 one should run the following:

```
1 ./gradlew test
```

And, in order to run the profiling application mentioned in section 5, one should run:

```
1 ./gradlew run
```

3 Design & Implementation

3.1 The *Memory Managed Object* Model

At the heart of the implementation of the treadmill garbage collector is a generic system for representing objects which reside on a heap. It is the intention that MemoryManagedObjects are the layer working in the

JVM which sit *outside* the aforementioned heap; they make it very easy to abstract significant data over a heap. The *MemoryManagedObject* (see Java source code in ”object.management.MemoryManagedObject”) is a type which contains the following information:

- a reference to the heap the object is allocated on (null if unallocated)
- the position, ”address”, on the heap
- a list of properties associated with this object
- a size (this is calculated based on the list of properties)

Properties A memory managed object can be viewed as a window onto a heap, given a position and size. The properties specify what comes out of and what goes through the window (particularly also *where*). Properties have a relative address which is used to determine the offset from the memory managed object to read data from. Properties are completely specified by: a relative address; a size in heap words; a marshalling function; and an un-marshalling function.

A simple case of a property, for example, is the *IntProperty* (all base properties are defined in the package ”object.properties”). IntProperty extends the generic property type, Property<Integer>”; it is of size one, and it’s marshalling function simply casts an Integer into a heap word.

Class Properties There is a use for a property which marshalls and unmarshalls a Class type (see 3.3). These are implemented by using a serialized representation of the class they intend to marshall and unmarshall.

Reference Properties Reference properties are essential given what we are trying to implement. A reference property is a special case of an IntProperty, in that, we store an address using the regular IntProperty methods, but we also store an *instance* of the thing we are trying to reference.

In order to construct a memory managed object, one needs to call ”addProperty” within the constructor of the object. This is utilised in the presented implementation by keeping public properties on the Episcopal object types, and have them work like ”getters” and ”setters”.

3.2 Heap Allocator

Underlying the rest of the implementation of the treadmill garbage collector is a very simple heap allocator model (see the source file for ”gc.BasicAllocator”). The heap in this case is an array of Java *long*. This is also to say, the ”word size” of the heap, is that of a long in Java.

When we add an object to the heap, that object is added to a set stored outside the heap; the object gets a reference to the heap it is allocated on; and finally it is assigned an address. The affect of this step is that the heap knows about the object, and the object knows about the heap (and its own position on it).

The heap also stores a *free list*, so we can allocate into the first space that has enough space for the object. Free region coalescing is not implemented, however this is a little outside the scope of what is trying to be achieved in the implementation.

3.3 Episcopal Objects

Given the subsequent explanation of memory managed objects (see 3.1), we can move onto the *Episcopal* objects required of the specification.

Distributions and Functions Imagining the object representation was to be used by a language designer working on the JVM (as the implementation language is Java in this case). It seemed fitting that the Distrib, PDistrib and Function types would hold an item representing a class itself. The class is marshalled and unmarshalled through a simple . It seems sufficient to do this, so that in the wider implementation of the *Episcopal* language one would use some form of reflection to instance the distribution or function (which may defined as a class file compiled from user code, for instance). Details of how distributions and functions work are deferred to the interfaces (presently empty), defined in the abstract classes in "episcopal.representations".

Other Episcopal object types are fairly straight-forwardly implemented as defined in the specification given the building blocks outlined previously.

3.4 Treadmill Garbage Collector

The implementation Baker's treadmill is more or less as it is described in his original paper (1). There are perhaps subtle differences in how the allocation to the heap itself works. Also, one would expect that Baker's implementation deals with a tag-based model for following links of an object on the heap - whereas in this case, objects are instanced directly in the JVM and work like "windows" onto the heap as described previously, so there is no need for tags (unfortunately, these mere windows are also garbage collected by the JVM itself, outwith the control of the code user).

The entirety of the treadmill system itself resides in "gc.TreadmillAllocator". It's worth noting what happens in the "flip" function. As outlined by Baker himself, "ecru" nodes are all turned white, and all black nodes are turned to ecru (marked as potential garbage for the next round of scanning). In the presented implementation, new white nodes are also added in between these steps if there are no white nodes. The idea is that: after calling flip there should be an available node; if no node is available after flipping, an exception is thrown, something has gone wrong (see the "allocate" function to observe this).

4 Testing

Given that the final garbage collection implementation is built on top of so many other components, effort was made to test these components as well. Unit tests have been provided which can be run as shown in section 2.

The treadmill itself is tested in "gc.TreadmillAllocatorTest", we show an example of allocating a function, the function's arguments as pointers, as well as the values they point to. We then observe in the test that when the argument pointers (Indirects) themselves are freed, their values are marked unreachable and are also freed.

If one wants to see this working, one should change the TreadmillAllocatorTest's instance of TreadmillAllocator to have "DebugMode.BASIC" rather than "DebugMode.NONE" as it is in the presented solution. If we do this, we see debug output (an ASCII representation of the treadmill's doubly-linked list) like that which is shown in figures 1 and 2.

```
_____________ after allocation of episcopal.Indirect@4fa66fa8 _____________
[/B/: ECRU: episcopal.Indirect@10cb3a5c] <==\
/=========================================/
\==> [/T/: GREY: episcopal.Function@73eaab2f] <==\
/=================================================/
\==> [/S/: BLACK: episcopal.Indirect@4fa66fa8] <====> /B/
```

Figure 1: debug output after adding a function two indirect objects - note the function is marked grey because it is a root node

```
_____________ after freeing object episcopal.Indirect@6a2b8644 _____________
[/T/: GREY: episcopal.Function@73eaab2f] <==\
/=========================================/
\==> [/F/: WHITE: NULL] <==\
/=========================/
\==> [/: WHITE: NULL] <==\
/========================/
\==> [/: WHITE: NULL] <==\
/========================/
\==> [/: WHITE: NULL] <==\
/========================/
\==> [/: WHITE: NULL] <==\
/========================/
\==> [/: WHITE: NULL] <==\
/========================/
\==> [/: WHITE: NULL] <==\
/========================/
\==> [/: WHITE: NULL] <====> /T/
```

Figure 2: debug output after adding a function and four indirects, each pointing to an allocated integer; we then free the indirects, and observe that the ints they were pointing to are also freed (there are eight white nodes)

5 Analysis

Analysis Setup For analysis of time complexity of the collection step itself, I've chosen to observe time taken to garbage collect with the depth of indirection. This is done in the simplest possible case, by linking Episcopal object "Indirects", one after the other in a chain. When the pointer pointed to by the root Indirect is freed, we expect all of the indirects reachable from it to also be freed.

Profiling results for inserting new nodes is also provided, we measure the time against the number of objects being allocated.

The profiling is run by the code in the package "profiling", which averages out the profiling time over a number of runs. The data generated by the profiler is present in the "data" directory.

Freeing Deep Indirects Analysis reveals what we'd expect of the garbage collection (mostly just the flip) phase in terms of time complexity. As shown by figure 3. When the object is freed, all the now-unreachable objects are also freed. Freeing an object is of $O(1)$, and we do this for a number of times proportional the number of objects that are no longer reachable (at most $n - 2$); hence the garbage collection phase appears as $O(n)$ in the worst-case, where n is the number of objects on the heap.

Allocating New Objects It appears that there is some inefficiency in allocating new objects as shown in figure 4. It looks as though the of change in time taken increases with the amount of objects being allocated. This cannot be taken as fact without further analysing for much more objects, however, and performing proper regression analysis to gain confidence in this theory.

It's not certain what's causing this inefficiency, as one would expect the operation of allocating for a new object to be of $O(1)$. As Java is the host language, we could speculate about what the JVM is doing in the background, but more work would need to be done to be sure of this.

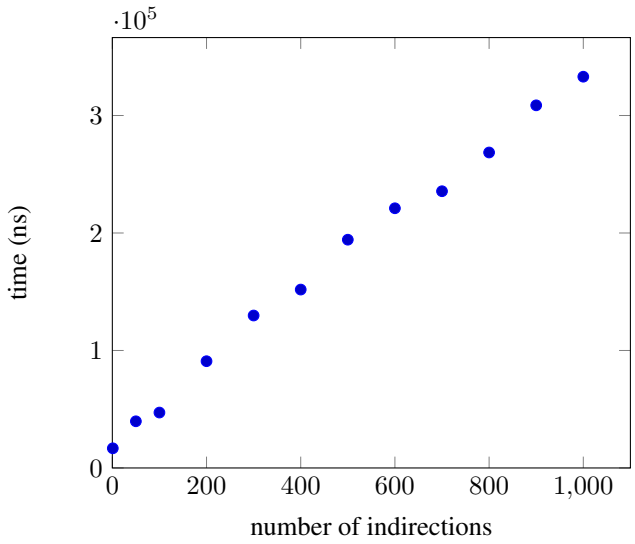

Figure 3: time taken to complete free with number of connected indirections

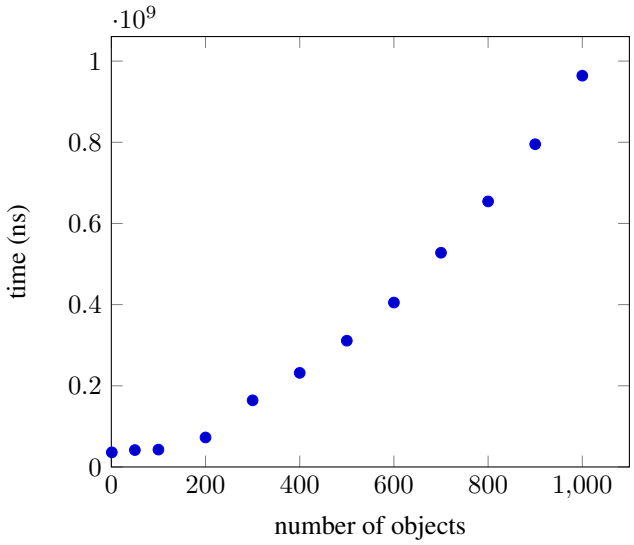

Figure 4: time taken to allocate with the number of objects

6 Evaluation

The presented solution, while maybe exhibiting symptoms of *chronic inefficiency* in some places, is a very solid, tested base for rectifying these inefficiencies. The *Memory Managed Object* system was designed with language designers in mind, and appears very easy to extend. Possible further work, for example, include the free region coalescing mentioned in section 3.2; as well any holes in the analysis shown in section 5.

Also provided in the solution is an analysis which provides further evidence for Baker's treadmill to have $O(n)$ collection time, for n objects in the heap.

References

[1] Baker, H. G. (1992). The treadmill: Real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70.