Garbage Collection Tutorial

The Unreachables

In languages such as C and C++, objects which we create at runtime (e.g. using `malloc()` or `new`) have to be explicitly deleted. Memory leaks are the result of objects being allocated, getting lost and never freed. It is possible to write memory tracking systems which allocate memory for us and know when we no longer need it. Some languages have this feature built in, including Lua.

There are various ways of tracking which memory objects are no longer needed. Once none of an applications variables reference an object it is said to be *unreachable*. Objects which are no longer reachable become deleteable and are candidates for automatic disposal, or *garbage collection*.

In the following examples we create some Lua objects and assign them to a variables. If we assign other values to the variables, and no other variable references the previous objects, they become *unreachable*. If it were not for garbage collection this would cause a memory leak. E.g.,

```
> t = { this="table"; 1,2,3 }  -- construct a table and assign it to "t"
> t = nil                      -- set "t" to nil and the table becomes unreachable
>
> s = "a string"               -- create string variable
> s = "another string"         -- when set to another value the old value becomes unreachable
```

Simple collection algorithms

There are various types of collectors and a lot of terminology associated with garbage collection [1]. One of the simplest collection algorithms is *reference counting*. Here we just keep a count in each object allocated, of the number of objects referencing (or using) it. If the number of objects referencing another object falls to zero it becomes unreachable (and unused) and can be freed. A slight complication with this is *cyclic referencing*, where object A references object B, and B references A. This is fine as long as other objects reference A and/or B, but when they no longer do, A and B will form an *island*. They are now a group of objects which are unreachable and mutually permanent because of cyclic referencing. The island will never be freed unless the reference counting collection algorithm is expanded. E.g.,

```
> a = {}
> b = {}
> a['other'] = b
> b['other'] = a    -- now we have a cyclic reference
> a,b = nil,nil     -- oh dear, we don't have a reference to the 2 tables now
```

```
> -- there are two tables which are unreachable
```

Mark and sweep

To get rid of islands from the algorithm described above we could travel through all of the variables in our system and see which objects they reference. If the variables find certain objects unreachable we can garbage collect them. This algorithm is called *mark and sweep*, i.e., we mark all the reachable objects and sweep away the ones remaining. Lua uses the mark and sweep garbage collection algorithm exclusively. This has the advantage that we don't have to reference count objects and so don't get cyclic referencing problems. A disadvantage is that the algorithm takes time to process and can be an issue in realtime applications.

Note: If we were to distribute the processing of the mark and sweep collection algorithm, so that it was not all done at once, it would be more transparent to the host system. This is called *incremental garbage collection*. This change is planned for Lua version 5.1.

Control and status

Lua provides some functions for controlling and querying the status of garbage collection. There are more details in the CoreFunctionsTutorial.

- `collectgarbage([limit])` sets the threshold at which garbage collection is initiated. It can be called without arguments to force an immediate collection.
- `gcinfo()` returns two numbers: the amount of memory that Lua has allocated and the current garbage collector threshold. Both are in Kbytes.

More reading

Other pages on the wiki cover garbage collection in Lua in more detail. The following are suggested reading:

- GarbageCollection - General links and information on garbage collection topics.
- OptimisingGarbageCollection - Notes on minimising garbage collection and the processing involved, should this be an issue. You might also want to read OptimisationTips.
- GarbageCollectionInRealTimeGames - More notes on garbage collection in Lua specific to games programming.