The Teckel Cyclic Reference Count Garbage Collector

Dick Grune
dick@dickgrune.com

October 23, 2014; DRAFT

° ° probaly obsolete

1 Introduction

The initial form of a Teckel program is a pure binary tree, as delivered by the parser. Several processes may turn it into a dag: common subexpression elimination; lazy copying; parameter substitution; and perhaps others. When the teckel is at rest, only three locations can point to tree nodes: the `Root`, the name list, and pointers in tree nodes. When the teckel runs, pointers are only consulted, not modified; and when it acts on `Node`, very few pointers are kept in local variables, and the pointer situation is easily controlled. Reference counting garbage collection (RC) is perfectly suited for such dags.

One or two processes might benefit from allowing a cycle in the dag, thereby formally turning it into a full-fledged graph. One such process may be the evaluation of a generated set, where some nodes may need access to earlier nodes to check uniqueness; another may be the linked name lists of nested routines, which point backwards in the dag.

Perhaps° both problems can be solved without creating cycles in the dag, but cycles may not be as much of a problem as is generally thought. So it will be interesting to see whether RC can be extended to a cyclic RC (CRC) capable of handling cycles, and how far the reference counts already present in the nodes may help. ° almost certainly

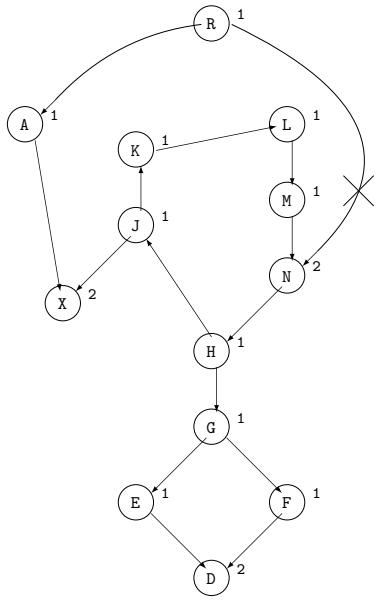
2 The Algorithm

Suppose the program contains a cycle, as in Figure 1, where the nodes have been given arbitrary names, and the reference counts are shown next to the nodes; initially the reference count of the node marked `N` is 2.

Now suppose the teckel decides to release[1] node `N`. The refcount of `N` sinks to 1; in the normal operation of RC the node cannot be freed, and 10 nodes are lost, which could be freed.

[1] A node is "released" when a pointer to it is given up. A node is "freed" when it is returned to the memory allocator (`Malloc`).

Figure 1: Imminent removal of a cyclic structure



This raises the question under which condition(s) a node `N` *can* be freed when it is released; this question has a fairly simple answer: when all nodes directly pointing to it can be reached only through `N` (note that having a refcount of 0 is a special case of this condition).

The nodes that can be reached through `N`, the set $\mathcal{A}$, can easily be found by depth-first visit, but it turns out that we can also find the ones that can *only* be reached through `N`. To this end we allocate an access counter, `access_count`, in each node, initialized to 0. We then perform a depth-first visit, V_1, from `N`, incrementing the `access_count` for each node we visit. The result is shown in Figure 2 where the access counts are is given between square brackets. Now nodes that can be accessed directly by nodes that cannot be reached by `N` can be recognized by `refcount`>`access_count`; we call this set $\mathcal{D}$. This set is not computed explicitly, but its members are recognizable.

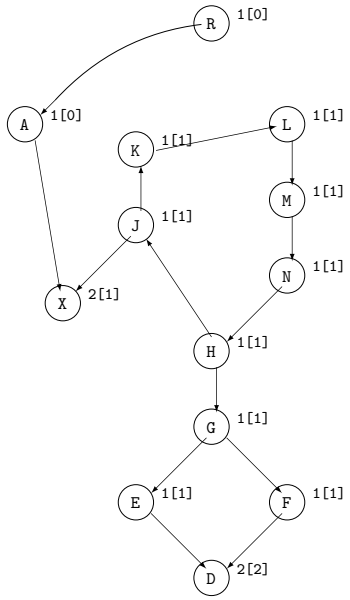
If `N` itself is in $\mathcal{D}$, `N` cannot be freed, since there is at least one node directly pointing to it that cannot be reached from `N`. And if it cannot be reached from `N`, it certainly cannot be reached *only* from `N`.

So if $\texttt{N} \in \mathcal{D}$ the problem is solved, but if $\texttt{N} \notin \mathcal{D}$ the problem is not solved, since it still has to be established whether any of the other nodes in $\mathcal{D}$ can indirectly access `N`.

The nodes in $\mathcal{D}$ can be found by another depth-first visit, V_2, from `N`; then for each such node we do a depth-first search for `N`, and if any can reach `N`, `N` cannot be freed.

This cannot be implemented as it stands, since the Teckel graph visiting mechanism is not reentrant. We therefore modify the graph visit V_1 to also mark all nodes that can reach `N` with a marker **R**; this can conveniently be done in its post-visit phase. Now V_2 finds all nodes in $\mathcal{D}$, and if any has the marker

Figure 2: A cyclic structure with access counts



R set, it is a "show stopper", and `N` cannot be freed.

If there is no show stopper, `N` can be freed, but this requires some care. Referring to Figure 2, we see that releasing its children causes a chain of events, first releasing `H`, `G`, `E`, and `D`, then freeing `E`, etc., until `M` is released, which releases its child `N`. Since `N` has no children any more (`H` just having been released), `N` itself is freed as any other node, `M` is freed, until at last `H` is freed, after which finally `N` is freed, for the second time!

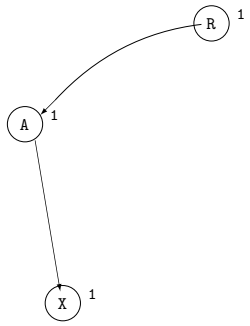
To prevent `N` from being freed by being released by `M`, we use a trick: before releasing the children of `N` we increase the refcount of `N` by one. At the end of the operation `N`'s refcount is decremented by 1; it should come out 0, and `N` can be freed. The resulting tree is shown in Figure 3. Note that CRC removes the the set $\mathcal{A}\backslash\mathcal{D}$.

Incrementing by 1 is sufficient even if there are multiple references to `N`: suppose there are k references to `N`, then its refcount is k, and $k+1$ after incrementing. Removing $\mathcal{A}\backslash\mathcal{D}$ removes the k references, and a refcount of 1 remains.

3 Efficiency

Since cycles occur sparingly if at all in Teckel, one would hope that the efficiency of CRC would be a minor issue. However, when CRC is called and recovers no nodes, either because there was no cycle or because there was a show stopper, its effort is completely lost. So the efficiency of the use of CRC is increased by knowing which nodes are on a cycle. Fortunately it is (almost?) always known when the cycle is made. The nodes involved can then be marked, and CRC be

Figure 3: The result of releasing node `N`



invoked only on nodes marked as cyclic.

The complexity of CRC is probably linear. The worst case cannot occur: one cannot make a clique of k binary nodes, since there are not enough pointers to do so. The most complicated case I can think of is a cycle of k nodes, each pointing to its successor and predecessor. Such a chain would require $k-1$ calls of CRC to free, but each call would free 1 node and run in constant time. A proof (or counterexample) would be nice.

Failing to call CRC on a removable cycle causes the immediate loss of the set $\mathcal{A}\backslash\mathcal{D}$, but the set $\mathcal{D}$ will eventually be lost too, since its refcounts remain too high. This can be seen by considering Figure 2. When CRC is not called, `J` will not be released, and `H`'s refcount will not be adjusted correctly, which will prevent it from eventually being freed. In addition the incorrect refcount will be flagged by `check_refcounts()`, a checking routine which is called regularly to prevent errors from developing.

4 Evaluation

Teckel's CRC is a full garbage collector that will correctly remove all inaccessible nodes from any binary graph. It is somewhat related to (but still quite different from) Christopher's garbage collector.[2]

That said, it must be noted that its indiscriminate application upon any pointer assignment is much too expensive, since it could easily cause the whole graph to be scanned each time. So a calling criterion must be defined.

Its application in Teckel is (probably) warranted, due to the fact that cycles are rare in Teckel and their construction is always(?) noted.

[2] T.W. Christopher, *Reference count garbage collection*, SP&E, **14**,(6),1984, pp. 503-507