CYCLIC REFERENCE COUNTING WITH LOCAL MARK-SCAN

Alejandro D. MARTÍNEZ and Rosita WACHENCHAUSER
Escuela Superior Latinoamericana de Informática, Pereira Irigol CC 3193, 1000 Buenos Aires, Argentina

Rafael D. LINS
Departamento de Informática, Universidade Federal de Pernambuco, Av. Prof. Luiz Freire, 50.000 Recife, PE, Brazil

Communicated by P. Henderson
Received July 1989

Keywords: Compilers, analysis of algorithms, functional programming, garbage collection

Introduction

The process of reclaiming unused storage space is called garbage collection. The two most simple algorithms used for garbage collection are mark-scan and reference counting.

The mark-scan garbage collection algorithm works in two phases. If a machine runs out of space, the computation process is suspended and the garbage collection algorithm is called. First, the algorithm traverses all the data structures (or cells) in use putting a mark in each cell visited. Then the scan process takes place collecting all unmarked cells in a free-list. When the mark-scan process has finished, computation is resumed. The amount of time taken for garbage collection by the mark-scan algorithm is proportional to the size of the heap (the work space where cells are allocated). The copying algorithm is a modified version of the mark-scan algorithm in which the heap is divided in two halves. The algorithm copies cells from one half to the other during collection. Its time complexity is proportional to the size of the graph in use. Mark-scan and copying algorithms generally traverse all the reachable data structures during garbage collection, which makes them unsuitable for real-time or large-virtual-memory applications.

In reference counting, each data structure or cell has an additional field which counts the number of references to it, i.e., the number of pointers to it. During computation, alterations to the data structure imply changes to the connectivity of the graph and, consequently, readjustment of the value of the count of the cells involved. Reference counting has the major advantage of being performed in small steps interleaved with computation, while other algorithms imply suspending computation for much longer. The disadvantage of the trivial algorithm for reference counting is the inability to reclaim cyclic structures. To solve this problem, a mixture of mark-scan and reference counting has already been used in the past. We refer to [3] for a detailed analysis of these algorithms.

More recently, several garbage collection algorithms have been developed for cyclic structures. In [4] Friedman and Wise present an algorithm which can recover cyclic structures that are created in one operation, and never modified thereafter. This is the case for the cyclic representation of recursive functions in Lisp and functional languages. Bobrow [1] gives an algorithm which can, in principle, recover all cyclic structures. His method relies on explicit information provided by the programmer. Bobrow collects nodes of the graph together to form groups and associates a reference count with a group rather than an individual data structure. Hughes’ algorithm [5] is based on Bobrow’s. It has the major
advantage of not needing extra information provided by the programmer. Hughes' algorithm is suitable for the implementation of functional languages such as Miranda [9]. Another algorithm for cyclic reference counting was presented by Brownbridge in [2]. Brownbridge's algorithm classifies pointers as strong and weak. Weak pointers, in general, close cycles. Brownbridge's algorithm was developed with general aims. However, as explained in Salkild's thesis [7], Brownbridge's algorithm is not correct. A new algorithm based on Brownbridge's algorithm is presented in [6].

In what follows, we present a general garbage collection algorithm based on reference counting, which deals with cyclic data structures, and prove its correctness.

1. The algorithm

The general idea of the algorithm here is to perform a local mark-scan whenever a pointer to a shared structure is deleted. The algorithm works in three phases. In the first phase we scan the graph below the deleted pointer, rearranging counts due to internal references and marking the nodes as possible garbage. In phase two, the graph is rescanned and any subgraphs with external references are marked as ordinary cells, and their counts reset. All other nodes are marked as garbage. Finally, in phase three all garbage cells are collected and returned to the free list.

Let us present implementation details of this algorithm. As usual, we link together free cells in a structure called a free-list. We say a cell $A$ is connected to a cell $B$ ($A \rightarrow B$) if and only if there is a pointer $\langle A, B \rangle$. We say a cell $A$ is transitively connected to a cell $B$ ($A \leftarrow B$) if and only if there is a chain of pointers from $A$ to $B$. The initial point of the graph to which all cells in use are transitively connected is called root. In addition to the information of number of references to a cell, we use an extra field which keeps the colour of the cell.

There are three operations on the graph:

- **New(R)** which gets a cell $U$ from free-list and creates the pointer $\langle R, U \rangle$, where $R$ is a cell transitively connected to root.

- **Copy(R, $\langle S, T \rangle$)** which creates the pointer $\langle R, T \rangle$, where $R$ and $S$ are cells transitively connected to root and there exists the pointer $\langle S, T \rangle$, and

- **Delete($\langle R, S \rangle$)** which deletes the pointer $\langle R, S \rangle$ from the graph and readjusts the graph.

These operations can be described algorithmically as follows:

\[
\text{New}(R) = \\
\text{select } U \text{ from free-list} \\
\text{set } RC(U) := 1 \\
\text{make pointer } \langle R, U \rangle \\
\]

\[
\text{Copy}(R, \langle S, T \rangle) = \\
\text{make pointer } \langle R, T \rangle \\
\text{increment } RC(T) \\
\]

\[
\text{Delete}(\langle R, S \rangle) = \\
\text{decrement } RC(S) \\
\text{remove } \langle R, S \rangle \\
\text{if } RC(S) = 0 \text{ then} \\
\text{for } T \text{ in Sons}(S) \text{ do} \\
\text{Delete}(\langle S, T \rangle) \\
\text{link } S \text{ to free-list} \\
\text{else} \\
\text{markred}(S) \\
\text{scan}(S) \\
\text{collectblue}(S) \\
\]

A cell $T$ belongs to the set Sons($S$) if and only if there is a pointer $\langle S, T \rangle$. markred is an auxiliary function which paints all the cells in the subgraph below its calling point as red. This indicates that these cells may be garbage. It also decrements the reference counts of the cells visited, leaving only counts which refer to pointers external to the subgraph. These operations are performed as

\[
\text{markred}(S) = \\
\text{if colour}(S) \text{ is green then} \\
\text{set colour}(S) := \text{red} \\
\text{for } T \text{ in Sons}(S) \text{ do} \\
\text{decrement } RC(T) \\
\text{markred}(T) \\
\]
scan searches the subgraph (now painted red) for external references to it. If during scan an external reference is found (a cell has an external reference if its reference count is greater than zero, due to marked), an auxiliary function is called to paint green the transitive closure of the external reference. Cells with no external references are painted blue.

\[
\text{scan}(S) =
\begin{cases}
\text{if colour}(S) \text{ is red then} \\
\text{if RC}(S) > 0 \text{ then} \\
\quad \text{scanred}(S) \\
\text{else} \\
\quad \text{set colour}(S) := \text{blue} \\
\quad \text{for } T \text{ in Sons}(S) \text{ do} \\
\quad \quad \text{scan}(T)
\end{cases}
\]

\text{scangreen}(S) =
\begin{cases}
\text{set colour}(S) := \text{green} \\
\text{for } T \text{ in Sons}(S) \text{ do} \\
\quad \text{increment RC}(T) \\
\quad \text{if colour}(T) \text{ is not green then} \\
\quad \quad \text{scangreen}(T)
\end{cases}

\text{collectblue} \text{ recovers all the blue cells in the subgraph below its calling point (garbage) and links them to the free-list.}

\text{collectblue}(S) =
\begin{cases}
\text{if colour}(S) \text{ is blue then} \\
\quad \text{set colour}(S) := \text{green} \\
\quad \text{for } T \text{ in Sons}(S) \text{ do} \\
\quad \quad \text{collectblue}(T) \\
\quad \text{remove } \langle S, T \rangle \\
\quad \text{link } S \text{ to free_list}
\end{cases}

2. Examples

Since the operations of New and Copy are the same as in the standard algorithm we will draw our attention only to the case of Delete.

We will show the way the algorithm works in the case of a cyclic graph in which the pointer deleted does not isolate the cycle, and a cyclic graph in which the pointer deleted isolates the cycle from root and causes the cyclic subgraph to be collected.

2.1. Cyclic graph preserved

In this example, the deletion of a pointer will not cause the recycling of any cell, because of sharing. Consider

```
root(G, 0)  free-list
\quad \varepsilon(G, 2) \quad \alpha(G, 1)
\quad \delta(G, 2) \quad \beta(G, 2) \quad \text{nil}
```

and the deletion of pointer \( \langle a, b \rangle \). marked is then called on \( b \), which is a shared cell. This transformation yields the following graph:

```
root(G, 0)  free-list
\quad \varepsilon(R, 1) \quad \alpha(G, 1) \quad \beta(R, 0) \quad \text{nil}
```

Now scan is called recursively to check if this subgraph is completely isolated from root. As soon as it reaches \( c \), an external reference is found which causes scangreen to be called on \( c \). Before scangreen is executed the graph looks like

```
root(G, 0)  free-list
\quad \varepsilon(R, 1) \quad \delta(B, 0) \quad \beta(B, 0) \quad \text{nil}
```

and after its execution, we have

```
root(G, 0)  free-list
\quad \varepsilon(R, 1) \quad \delta(B, 0) \quad \alpha(B, 1) \quad \beta(B, 0) \quad \text{nil}
```

Now, collectblue is called recursively from \( b \). The graph will be kept unchanged due to the absence of blue cells.
2.2. Cyclic subgraph collected

In this example we present the case of a graph in which the deletion of a pointer will cause the isolation of a cycle and therefore its collection. If we have the graph

we will delete pointer \( \langle \text{root}, a \rangle \) from the graph above. Since \( a \) is a shared cell, a call to \textit{markred}(a) will take place, yielding

Now \textit{scan} will be applied recursively to the graph starting at \( a \), looking for cells with external references. The only external reference to the subgraph below \( a \) is found in \( d \). This will provoke a call to \textit{scangreen} on \( d \). Just before this call, we have

\( \text{collectblue} \) is called on \( a \), and the whole cycle below \( a \) is collected, giving as result the following structures:

Note that the subgraph below \( d \) was transitive connected to \( \text{root} \) through a path that does not involve the deleted pointer \( \langle \text{root}, a \rangle \), and thus it is preserved.

3. Proof of correctness

We now present an informal proof of the correctness of the algorithm.

3.1. Basic condition

Every cell is either (transitively) connected to the root or is in the free list, but not in both.

3.2. The basic condition is always true provided that there is no active call to \textit{Delete}

3.3. The operations preserve the basic condition

- \textit{New}: \textit{New} removes a cell \( U \) from \textit{free-list} and transitive connects it to \( \text{root} \). The remaining cells in the \textit{free-list} remain as such, satisfying property 3.2 above and clearly meets the basic condition. The status of other cells remain unchanged.
- \textit{Copy}: \textit{Copy} makes a new link to an already existing cell which is transitively connected to \( \text{root} \). \textit{Copy} is only applicable to cells transitively connected to \( \text{root} \), therefore preserving property 3.1.
- \textit{Delete} has two cases:

  Case 1. If the pointer deleted is the last pointer to a cell \( S \) this will cause the reference count of \( S \) to drop to zero \( (RC(S) = 0) \). If \( S \) has no sons it will be connected to \textit{free-list}, preserving the basic
condition. If $S$ has sons, call Delete on each of them and then connect $S$ to free-list. Since the sons of $S$ are accessible by active calls of Delete and $S$ will be linked to the free-list after the termination of the calls to its sons, property 3.2 holds at each recursive call of Delete.

Case 2. If the pointer is not the last pointer to a cell $S$, this may mean that it was the connection between a cycle and root (this is the case in which the standard reference count algorithm fails to recycle garbage). The algorithm will then perform a local mark-scan on the cells below $S$. First, all cells below $S$ are painted red, decrementing the reference count of cells in order to ignore references internal to the subgraph below $S$ (by recursive calls of markred). This subgraph is then scanned (by recursive calls of scan) to search for external references to the subgraph below $S$, and the cells with no external references are painted blue, until an external reference is found. In this case, the whole subgraph below this external reference is in use and should be painted green (by recursive calls of scangreen), because they are transitively connected to root via the external reference in the graph below $S$. If after scangreen there are still cells painted blue below $S$ this means that these cells are unreachable from root. The cells painted blue will be collected by recursive calls of collectblue, placing them on the free-list. Cells not painted blue remain connected to root.

4. Conclusions

The algorithm presented here is a simple reference counting garbage collection algorithm for cyclic structures, which works as a natural extension of the standard reference counting algorithm. Depending on its application the cost of the use of this algorithm may be extremely low.

In the implementation of functional languages, for example, it is a well-known fact that most structures have reference count one [8]. The cost of the use of this algorithm in this case would usually be exactly the same as the standard reference count algorithm.

Deletion of a pointer to a shared structure increases the complexity of the local mark-scan algorithm to $O(n)$, where $n$ is the size of the shared subgraph.

Acknowledgment

The authors are grateful to Silvio Meira and Manoel Lemos of U.F.P.E. (Recife, Brazil), and Simon Thompson and John Cupitt of U.K.C. (Canterbury, UK) for their comments.

This work was partly supported by Project ETHOS of the Argentinian–Brazilian Programme for Research and Advanced Studies in Informatics, C.N.Pq. (Brazil) grant No. 304498/87-4/CC, and British Council and C.N.Pq. (Proc. 910329/88-5/CC) HED link.

References