On-the-fly garbage collection for several mutators

Jan E. Jonker

Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, NL-9700 AV Groningen, The Netherlands

Received March 1990 / Accepted January 1992

Summary. An algorithm is given for on-the-fly garbage collection in the presence of several mutators. It uses two colours and is a generalization of Ben-Ari's algorithm (1984). The correctness proof is based on the lexical orderings of several tuples of state space functions. It is shown that in a certain sense the algorithm is optimal. Three variations of the algorithm are given and proved correct. In the case that there is only one mutator one of these variations closely resembles a well-known incorrect algorithm.

Key words: Concurrent programming – Correctness – Garbage collection

1 Introduction

In [2] the authors develop an algorithm for on-the-fly garbage collection. In [1] Ben-Ari presents a slightly simpler algorithm. The correctness proof of these algorithms is notoriously difficult and has been an object of study in itself. In [6] Van de Snepscheut develops a proof for Ben-Ari's version, using the Owicki-Gries theory as presented in [3, 5].

The authors of [1, 2, 6] consider garbage collection in the presence of one mutator only. In [4] a garbage collection algorithm in the presence of several mutators is given. In this paper we present a multi-mutator algorithm that is finer-grained, uses less colours and has less overhead for the mutators. We prove its correctness by the analysis of some judiciously chosen variant functions. Several variations of the algorithm are considered: the order of the mutator actions may be reversed, and/or the counting of marked nodes may be dispensed with. We present a correctness proof of these versions. We also consider the worst-case performance of the algorithms. If there is more than one mutator the version with interchanged mutator actions and no counting of marked nodes turns out to have the best worst-case performance.

2 Problem statement

We generalize the recapitulation of the problem in [6] for the case of several mutators:

"We consider a finite directed graph of varying structure but with a fixed set of nodes, in which each node has a fixed set of outgoing edges. In this graph a fixed subset of nodes exists, called the roots. A node is called accessible if a directed path exists along the edges from at least one root to that node, and it is called garbage otherwise. The program consists of a fixed, finite and nonempty set of mutator processes and one collector process. By considering both the head of the free list and the special node NIL to be roots, any mutator can be described as the repeated execution of 'redirect an outgoing edge of an accessible node towards an already accessible node'. Redirection of an edge may turn its previous target and some or all of its descendants into garbage nodes. It is the task of the collector to identify these garbage nodes and to append them to the free list."

3 Correctness conditions

The following correctness conditions are essential to a solution:
● Safety condition: only garbage nodes are appended to the free list.
● Progress condition: any garbage node is eventually appended to the free list.

Besides these we follow [1] and [2] and impose the following neatness conditions:

● The largest indivisible action contains at most one read or write of a shared variable, and shared variables are small objects.
● The space overhead (compared to a program without garbage collection) is small.
● The time overhead for the mutators is small.

These neatness conditions are purposely vague and serve mainly to compare two programs. For example, we would prefer a program having nodes and edges as shared variables to a program having a graph as a shared variable. Also, we would prefer a program that attaches to each node a color from a palette of two or three colors to a program that attaches extra pointers to each node. Finally, as far as concerns garbage collection we prefer to shift the burden of the computation from the mutators to the collector.

4 Informal discussion

We recapitulate the solution of the single mutator case given in [1, 2, 6].

The garbage collector is a cyclic process. Starting in a state in which all nodes are unmarked it can be described as the alternating execution of a marking phase, in which all accessible nodes and possibly some garbage nodes are marked, and an appending phase, in which all unmarked nodes are appended to the free list and all marks are removed.

The marking phase first marks all roots. Then several sweeps across all edges are made. Whenever an edge from a marked source to an unmarked target is encountered, that target is marked. The only contribution of the mutator to the collector’s task consists of marking the new target of an edge after redirecting that edge. The marking phase ends when a non-marking sweep occurs, i.e. a sweep during which no marks are added either by the mutator or by the collector.

In [6] it is proved that no unmarked accessible node exists after a non-marking sweep. It is also proved that the mutator’s activity does not interfere with either phase of the collector, even if an atomic unit of action may contain at most one access to a shared node or edge. Thus the safety condition is met. It is also shown in [6] that the progress condition is met. Finally, it is shown how the sweeps, the counting of marked nodes and the appending phase can be implemented in accordance with the neatness conditions.

In [2] it is indicated that the operation “append a node to the free list” can be implemented as a sequence of graph operations of the type “redirect an outgoing edge of an accessible node”.

If several mutators are present the collector must be adapted. We will show that a modification of the termination condition of the marking phase suffices: the marking phase ends after \( \# M \) consecutive non-marking sweeps, where \( \# M \) is the number of mutators. We will show that the safety, progress and neatness conditions are met by the modified program.

5 Notation

Before we present the program, we introduce some notational conventions. Because of the length of the formulas in the formal correctness proof (cf. the appendix) we want a rather compact notation. For the program variables this is achieved as follows.

The graph itself and the sets of nodes, edges and mutators remain anonymous. For reference to individual nodes, edges and mutators we introduce index sets \( N \), \( E \) and \( M \), respectively. Index-valued variables are denoted by the corresponding lower case letters. A clumsy phrase like ‘the node with index equal to the value of \( n \)’ will be abbreviated as ‘the node \( n \)’. The mark of a node \( n \) is implemented as a boolean shared variable \( n.b \) which is true if the node is marked and false if it is unmarked. An edge (with index equal to the value of \( e \)) is implemented as a pair of shared variables consisting of a source node index and a target node index, which we denote by \( e.s \) and \( e.t \), respectively. The centered dot binds to the left. Thus \( e.s.b \) denotes the mark of the source node of edge \( e \). The reader who is familiar with the programming language Pascal may think of \( N \) and \( E \) as sets of pointers to records on the (anonymous) heap, and of \( s \), \( t \) and \( b \) as record field names. He would write \( e.s.b \) as \( e.s1.b \). The reader who wants to introduce the explicitly named arrays \( \text{Node} \) and \( \text{Edge} \) would write \( e.s.b \) as \( \text{Node}[\text{Edge}[e].s].b \). It is easily seen that each centered dot corresponds to one access to a shared variable.

We do not want to introduce any particular ordering of the nodes and edges. Therefore, in the program we use set operations like union (\( \cup \)) and number of (\( \# \)). Singleton sets are denoted by their single element without surrounding braces.

6 Program

The program is an adapted version of the program in [6].

Initially, all roots are marked. The rest of the graph is initialized arbitrarily.

The mutators are coded in accordance with their description in Sects. 2 and 4. Angled brackets surround indivisible actions. The indivisible actions have been labeled for later reference. Annotation is provided within braces.

The mutators redirect edges from accessible nodes to accessible nodes only. This is achieved by making the choice of a new target and the actual redirection into a single indivisible action. For the purpose of the

...