An Introduction to Garbage Collection

0. What is garbage collection?

Garbage collection is a storage management technique with the typical property that storage locations whose contents have become irrelevant are not explicitly deallocated by the process that used these locations. Instead, a dedicated process, called the Garbage Collector, is used to identify and "recycle" such storage locations. For the sake of clarity I shall call the process(es) using the storage locations the Computation Proper.

First and for all, it must be emphasized that the problem of designing a garbage collector admits of very many solutions, with very different performance characteristics, depending on the requirements imposed upon the installation as a whole. Aspects of the installation that are particularly relevant are:

- whether or not all storage segments have the same size.
- absence or presence of virtual storage.
- absence or presence of parallelism.
- absence or presence of real-time requirements.

It goes without saying that in an introduction like this one we cannot (and shall not) pay attention to all these aspects and solutions, nor can we do justice to all that has been published on this subject: a recent compilation [2] of publications contains no less than 281 entries! For a

more extensive overview of the subject we refer to [0], while [1] also provides interesting reading material.

A very important question to be answered is to what extent the activities of the Garbage Collector and the Computation Proper will be interleaved. One extreme of the spectrum is the case where both processes run truly concurrently on different processors, the other extreme of the spectrum is the case where the Garbage Collector is activated only when the Computation Proper comes to a grinding halt because the available storage space is exhausted. In this introduction we shall only consider the latter case, as being the simplest; you probably can imagine that more-concurrent solutions have better real-time properties, although at the expense of their performance, and you probably can also imagine that in concurrent solutions the two processes must not interfere in undesirable ways.

In any design of a garbage collector performance considerations lie at the heart of the matter. I shall devote a separate section to this but I wish to stress here already that performance comes in two flavours: processor utilisation —speed— and storage utilisation —compactness—. The crux of the matter is —I will explain this later— that the, by themselves legitimate, desires for high speed and high compactness are mutually conflicting: you can have either but you cannot have both. Therefore, any balanced design of a garbage collector can at best attain a compromise

and yield an atmost reasonable speed as long as storage utilisation remains under an at most reasonable maximum.

Garbage collection is required for the implementation of functional programming languages and of object oriented languages. Remember, however, that garbage collection is not a language concept, but an implementation technique: strictly speaking, saying that language X is a "language with garbage collection" is as nonsensical as saying that functional programming language Y is a "lazy language".

[0] J. Cohen, Garbage Collection of Linked Datastructures, Comp. Surveys 13, nr. 3 (1981), pp. 341-367.

[1] S.L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall (1987).

[2] N. Sankaran, A Bibliography on Garbage Collection and Related Topics, ACM SIGPLAN Notices 29, nr. 9 (1994), pp. 149-158.

1. Specification of the problem

As already announced above we shall study the case in which the Garbage Collector is only activated when the Computation Proper runs out of free storage space. More precisely, as long as sufficient free space is available the Computation Proper allocates storage space in contiguous chunks called cells (or: segments); each cell is entirely identified by its address and its size. Different cells

may have different sizes; the problem becomes essentially simpler, and its solution more efficient, if all cells have the same size. Cells usually are small, say, from two to some tens of storage words; as a consequence, a (large) store can contain very many cells and any nontrivial computation will use very many cells.

The Computation Proper manages its collection of cells in use in the following way. The Computation Proper maintains a limited and well-identified set of variables containing pointers to cells. A pointer to a cell may be the address of that cell, but I prefer the (more abstract) notion of pointers because any mechanism that uniquely identifies cells will do. In what follows I shall abbreviate "pointer to a cell" to just "pointer": our pointers always point to cells, never to other things. Each cell may contain a mixture of data relevant to the Computation Proper but irrelevant to the Garbage Collector, and also pointers. A pointer in a cell may identify any allocated — there are no other ones — cell, even the cell containing the very pointer; thus, cells may contain pointers to themselves, and the cells and pointers may form arbitrary cyclic structures.

The Computation Proper is now assumed to obey the following

Important Rule: The Computation Proper only inspects (reads) or modifies (writes) the contents of a cell if a pointer to that cell is contained in one of the (abovementioned) variables.

□

If we call those cells accessible that are identified by the pointers in the variables then the rule states that only accessible cells may be inspected or modified. By reading a pointer from an accessible cell into a variable the cell identified by that pointer can be made accessible as well. By repeating this trick as often as necessary every cell that is identified by a chain of pointers can be made accessible. We call all such cells reachable and we conclude that, by following the chains, the Computation Proper can make every reachable cell accessible. More precisely, a cell is reachable if either a variable or another reachable cell (or both) contain a pointer to it.

Due to modifications in the contents of variables and/or cells, some cells may become unreachable. Strictly speaking, unreachable cells are still part of the state space of the Computation Proper, but unreachable cells can never be made accessible again; therefore, the future course of the computation does not depend on the contents of these cells, so these contents have become irrelevant and the storage space occupied by such cells may be safely deallocated and re-used for other purposes. In the current jargon unreachable cells are called garbage cells.

When the Computation Proper runs out of free space the whole store is filled with cells; every cell is either reachable or garbage (but not both). It is the task of the Garbage Collector to identify all garbage cells and administer them as free space. This is the minimum

requirement, to be met by every garbage collector. In addition, however, it may be necessary to collect all reachable nodes into one contiguous area, so that all free space is also collected into a single contiguous area. This is called compaction and is necessary to avoid that free space becomes fragmented into too many too small, hence useless, pieces. Only in the case that all cells have the same size (and in the absence of virtual storage) compaction will not be needed.

2. A global performance analysis

Most garbage collectors identify the garbage cells by a process of elimination: initially, all cells are "unmarked" and after all reachable cells have been "marked" the garbage cells are the ones that have remained "unmarked". The amount of time spent on the marking process will be (at best) proportional to the number of reachable cells, whereas the amount of time spent on compaction is, of course, proportional to the size of the whole store. So, we may safely assume that the time needed for a single run of the Garbage Collector is give by a formula of the shape:

$$\alpha \cdot x + \beta \cdot y \ ,$$

where α and β are parameters of the particular garbage collection strategy, and with:

x : the number of storage words occupied by reachable cells, and

y : the number of storage words occupied by garbage cells.

(So, $x+y$ is the size of the store.) Parameters α and β usually are nonnegative and may be expected to satisfy $\alpha \geq \beta$, but I shall not use this.

The yield of a run of the Garbage Collector is y words of free space, and because every word of free space thus recovered has been allocated once earlier, the average time spent on garbage collection per word allocated is:

(1) $$\alpha \cdot \frac{x}{y} + \beta .$$

Formula (1) is important because it gives the time overhead of garbage collection per allocation of one word of storage. How much this overhead really influences the speed of the Computation Proper also depends, of course, on the rate of allocation of cells, but this rate is not under our control. As such, formula (1) is a direct measure of the Garbage Collector's time performance —the lower the better—.

The space utilisation of the Computation Proper is the ratio of the amount of space occupied by reachable cells and the size of the whole store, that is, the space utilisation is the value η defined by:

$$\eta = \frac{x}{x+y} \quad .$$

By means of some elementary algebra formula (1) can be rewritten in terms of η as:

(2) $$\alpha \cdot \frac{\eta}{1-\eta} + \beta \quad .$$

This shows that the performance of a garbage collector depends critically on storage utilisation; when $\eta \to 1$ the overhead even becomes unboundedly large. This is no surprise: one run of the Garbage Collector takes quite some time and is, therefore, only effective if it yields quite some garbage, which requires that y is sufficiently large and η is sufficiently smaller than 1. A fair rule of thumb is that garbage collectors perform well as long as $\eta \leq 50\,\%$, but higher values of η are acceptable if the allocation rate is sufficiently low (or if speed is not important).

If we would wish to compare the performances of two different garbage collection strategies we should do so with the same storage utilisation. Because storage utilisation has to remain well below 1 anyhow, that is, because more storage space must be available than what the Computation Proper minimally needs, it may be worthwhile to use some of this additional space for administration by the Garbage Collector, thus possibly allowing for an increase in speed. Such a decision, to "waste" on administration what already

is scarce, may seem "counter-intuitive" [so what?], but if this indeed gives rise to a substantially faster garbage collector —having smaller values for α and β— the investment is well spent. The lesson to be learnt here is that design decisions should be based on a thorough analysis, not on "intuition"; moreover, the time performance of a garbage collector must always be understood in relation to storage utilisation.

3. Computing the reachable cells

Whenever we must design a nontrivial algorithm it is wise to ignore representation details as much as possible and to try to formulate the essence of the problem and its solution in abstract terms first.

In our case the structure of all allocated cells, reachable and garbage, together with the pointers forms a directed graph: every cell corresponds to one node in the graph, and for every two nodes x,y the graph has an arrow from x to y if and only if the cell (corresponding to node) x contains a pointer to the cell (corresponding to node) y. From Section 1 we recall that we called a cell accessible if (at least) one of the Computation Proper's variables contains a pointer to that cell. Here we shall call the nodes corresponding to accessible cells root nodes and we assume the root nodes to be known.

notation: x, y, z denote arbitrary nodes, boolean expression $x \to y$ means "the graph contains an arrow from x to y", and set A is the (given) set of root nodes.

□

Some of the nodes are reachable, others are not; so, "being reachable" is a predicate on the set of all nodes. Calling this predicate R we thus have as interpretation:

$$R.x \equiv \text{"node } x \text{ is reachable"}.$$

A node is reachable if it is a root node or if an arrow to this node exists from another reachable node. In formulae, R has the following properties:

(0) $\quad (\forall y :: y \in A \Rightarrow R.y)$

(1) $\quad (\forall x, y :: R.x \wedge x \to y \Rightarrow R.y)$

This is, however, not sufficient to fully define R. If we would choose, for example, $(\forall y :: R.y \equiv \text{true})$ this R would certainly satisfy (0) and (1), but in general not every node will be reachable. Put differently, (0) and (1) allow us to conclude reachability of some nodes, but we also need a rule by which nonreachability can be concluded. This rule is the following one, which states that $R.y$ is false unless (0) and (1) prescribe that $R.y$ is true:

(2) $\quad R$ is the strongest of all predicates satisfying (0) and (1).

The problem to be solved now is the construction of a program for the computation of R. For this purpose we introduce a variable S, of type "predicate on nodes" or, if you prefer, of type array node of boolean. (Keep in mind that we are designing an abstract algorithm: we shall decide upon a representation of the data later.) In terms of S the program's postcondition will be:

$$(\forall y :: S.y \equiv R.y) \text{ , or, as abbreviation:}$$

$$[\, S \equiv R \,] \quad .$$

(So, here I use a pair of square brackets $[\ldots]$ as an abbreviation for universal quantification over the set of all nodes.)

Because $[\, S \equiv R \,]$ is equivalent to $[\, S \Rightarrow R \,] \wedge [\, R \Rightarrow S \,]$ and because of (2), we can replace this postcondition by the conjunction of the following three requirements (imposed on S):

(3) $\quad [\, S \Rightarrow R \,] \quad$ (i.e.: $(\forall y :: S.y \Rightarrow R.y)$)

(4) $\quad (\forall y :: y \in A \Rightarrow S.y)$

(5) $\quad (\forall x,y :: S.x \wedge x \rightarrow y \Rightarrow S.y)$

There is no obvious initialisation for S that establishes all three conditions; in particular, (5) is awkward because S occurs in it on both sides of an implication. To enlarge our playing ground, and to weaken the postcondition, we now replace the leftmost S in (5) by a new variable T, of the same type as S and R. Thus we obtain what we shall use as the invariants for the program;

I have included an additional invariant $[T \Rightarrow S]$ (P_0), because this turns out to be handy:

P_0: $(\forall y :: T.y \Rightarrow S.y)$
P_1: $(\forall y :: S.y \Rightarrow R.y)$
P_2: $(\forall y :: y \in A \Rightarrow S.y)$
P_3: $(\forall x,y :: T.x \wedge x \rightarrow y \Rightarrow S.y)$

initialisation: T only occurs to the left of an $\Rightarrow$; hence, $(\forall y :: \neg T.y)$ is sufficient to establish P_0 and P_3. P_2 is established by $(\forall y :: S.y \equiv y \in A)$; with this precondition, P_1 becomes equivalent to $(\forall y :: y \in A \Rightarrow R.y)$, which is o.k. on account of R's definition, formula (0). The following program fragment is a correct initialisation:

for all y do T.y := false ; S.y := false od
; for all y ∈ A do S.y := true od

□

termination: We have obtained the invariants from the postconditions by replacing one occurrence of S by T. Therefore, the postconditions follow directly from the invariants whenever $T = S$. So, we shall use $T \neq S$, that is, $(\exists y :: T.y \neq S.y)$, as the guard for the repetition.

□

progress: By invariant P_0, the guard $(\exists y :: T.y \neq S.y)$ is equivalent to $(\exists y :: \neg T.y \wedge S.y)$. The set of all nodes being finite — the graph is finite — we can and shall use $(\#y :: \neg T.y)$ as the bound function. The

value of this bound function is nonnegative and is decreased by any assignment $T.z := true$ provided this assignment has precondition $\neg T.z$. Invariant P_0 requires that such assignment also has precondition $S.z$; fortunately, the guard guarantees the existence of such a z. Invariants P_1 and P_2 do not contain T but $T.z := true$ surely affects P_3. Therefore, the observations made thus far are done justice by a repetition of the following structure:

```
do T ≠ S → { (∃y :: ¬T.y ∧ S.y) }
             |[ var z : node
             ▷ z := "some value satisfying ¬T ∧ S"
             ; { ¬T.z ∧ S.z }
               T.z := true
             ; "restore P3"
             ]|
od ,
```

and the only obligation we are left with is to refine the phrase "restore P_3".

Invariant P_3 can also be written as:

P_3: $(\forall x :: T.x \Rightarrow Q.x)$, with Q defined by:

$$Q.x \equiv (\forall y :: x \rightarrow y \Rightarrow S.y) .$$

This shows that the only term in P_3 that is affected by $T.z := true$ is $Q.z$; so, "restore P_3" boils down to "establish $Q.z$". This is easy; the

annotation in the following program fragment shows that the assignment $S.y := true$ maintains the invariance of $P1$; notice that $S.y := true$ cannot violate $P0$, $P2$, or $P3$:

"restore $P3$":

{ $S.z$, hence by $P1$: $R.z$ }
forall $y : z \to y$ do { $R.z \wedge z \to y$, hence: $R.y$ }
$S.y := true$
od
{ $Q.z$, hence $P3$ }

In "$R.z \wedge z \to y$, hence: $R.y$" we have used formula (1) from the definition of R.

□

This concludes the construction of an abstract program for the computation of the reachable nodes.

* * *

The above program can be implemented in very many ways, depending on how the predicates S and T are represented. Here I will present just one of the many possibilities, without justifying my choice.

The assignment $S.y := true$ can be replaced by an equivalent construct, thus effectively strengthening the annotation for the assignment itself:

```
if S.y  →  skip
□ ¬S.y  →  { ¬S.y ∧ ¬T.y  (by P0) }
           S.y := true
           { S.y ∧ ¬T.y }
fi .
```

Both S and T can be represented by an array variable m, of type array node of node, and a variable t, of type node. Moreover, we need 2 dedicated constants, which I will call nil and nul, of type node, with the property that nil ≠ nul and that they differ from all nodes in the graph. Predicate S is represented by array m as given by the representation invariant:

P4: $(\forall y :: S.y \equiv m.y \neq \underline{nul})$

Moreover, all nodes y with $\neg T.y \wedge S.y$ are stringed together into a linear list, which is also represented by m and the variable t; the value nil is used to mark the end of this list and for every y, $y \neq \underline{nil}$, the first element of the list beginning at y is y itself and the remainder of this list is the list beginning at m.y:

P5: $(\forall y :: \neg T.y \wedge S.y \equiv y \in list.t)$, where

```
y ∈ list.x  ≡  if x = nil  →  false
               □ x ≠ nil  →  y = x ∨ y ∈ list.(m.x)
               fi
```

In terms of this representation the program can now be implemented as follows:

```
    t := nil
;   for all y do m.y := nul od
;   for all y∈A do  m.y := t ; t := y od
;   do t ≠ nil → ||[ var z : node
                   ▷  z := t
                   ;  t := m.t
                   ;  forall y : z → y
                      do if m.y ≠ nul  →  skip
                         [] m.y = nul  →  m.y := t ; t := y
                         fi
                      od
                   ]||
    od
    { (∀y :: "y is garbage"  ≡  m.y = nul ) } .
```

Variable m can be implemented as a separate array, as suggested by its type, but it can also be implemented by distributing its elements over the cells in the store; that is, within every cell y an additional word is allocated to hold the value of m.y. If cells are identified by pointers and if this additional word is named m then, in a Pascal-like notation, we would write y↑.m instead of m.y. Here we have a nice example of using storage space for administration: without this, garbage collection becomes much harder.

4. Compaction

Compaction is the process of moving all reachable cells to one end of the store, so that all garbage cells are melted together into one area of free space at the other end of the store. This makes addresses of cells volatile and the most difficult case arises when the pointers are just addresses: then all pointers must be adjusted to reflect the new situation. This can be done as follows.

The entire store is scanned twice, both times in the same direction. (This direction is irrelevant as long as the cells can be recognised.) During the first scan the addresses of the new locations of the reachable cells are calculated, but the cells are not yet moved; these addresses are stored in the additional words m of the cells. Moreover, all backward — relative to the scanning direction — pointers are adjusted. In between the two scans the root pointers (in the variables of the C.P.) must be adjusted. During the second scan the cells are moved to their new locations and all forward — relative to the scanning direction — pointers are adjusted; simultaneously, the m-fields can be reset to nul so as to reinitialise them for the next run of the Garbage Collector.

The number of scans can be reduced to 1 if pointers are not volatile addresses but location independent numbers. In this way, speed is gained

at the expense of even more administration (to map the numbers to addresses). In view of the analysis in Section 2 it is not immediately clear whether or not this is an improvement; in any particular case, a thorough analysis must be carried out to reveal this.

Similarly, when the sizes of the cells are, although not all the same, approximately equal it is an option worth pursuing to round up all sizes to their maximum value and to allocate cells of maximal size only. The resulting decrease in storage utilisation may be more than compensated by the increase in speed: compaction becomes superfluous and all that is needed is a simple scan to collect the addresses of the garbage cells into a free list. Again, only a thorough performance analysis can reveal whether this is an improvement.

5. Postscriptum

The above was mainly written for educational purposes, and doing so was a pleasure.

Eindhoven, 7 november 1995
Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology