CHAPTER 1

INTRODUCTION

Garbage collection, the automatic reclamation of dynamically allocated memory, is a mature memory-management technique. It has been in use since 1958 [McCarthy, 1960] and is now accepted as a customary element for dynamic memory management in run-time systems of modern programming languages, especially object-oriented ones, for reasons of safety and for its clear software engineering benefits [Gosling *et al.*, 1996]. The increasing acceptance of object-oriented languages makes it more important than ever for garbage collection to work as fast as possible.

Multi-generational copying collectors are the most popular memory reclamation algorithms in use today. Generational collectors have two real advantages over original non-generational collectors: pause times are shorter (on average), and total cost is lower.

There are, however, three tenets that are commonly held as explanations why generational collectors have these advantages, and that have discouraged exploration of alternative strategies. The first has to do with a property, empirically confirmed in object-based systems, that young objects die fast (more precisely, that object mortality is a decreasing function). The popular wisdom is that generational collectors exploit this property by concentrating effort on collecting young objects. Furthermore, whenever only a part of the heap is collected at once, the remainder must be assumed live for the purposes of the collection, and all pointers from outside the region collected into it are roots of the collection. This approximation is conservative, and considers more data live than is actually the case, and thus copying cost is greater. The second tenet is that this excess copying effort is small in generational collection because pointers from outside the region into it are, by construction, pointers from older objects to younger objects, and such pointers are few. The third tenet is that for the same reason it is easy to maintain the write barrier (actions taken during the execution of the program when a pointer is stored) and remembered sets (the sets of pointers crossing into a region).

Taken together, these three tenets do more than explain the good performance of generational collection. They have been understood in the past to rule out using any strategy that might collect older objects in preference to younger. Any such strategy would apparently concentrate effort on regions of lower mortality. It would also cause the collected region to have many incoming pointers from outside, thus from younger objects, and have both a high excess copying penalty and a high maintenance cost for the write barrier and remembered sets. Our study shows that, although each of the three tenets is intuitively and plausibly based on truthful observations of particular garbage collectors' behavior, alternatives to youngest-first generational collection are nevertheless feasible. The several plausible inferences drawn from the observations do not in fact rule out good incremental copying collectors other than generational: as we shall discover, they merely set the stage for a performance trade-off.

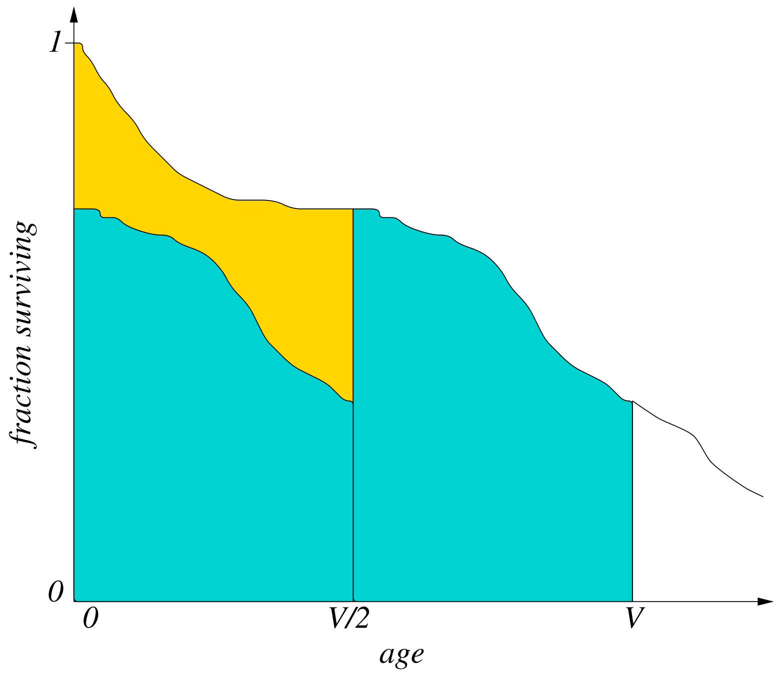

Figure 1.1. Surviving objects in a newly allocated area.

First, the intuition that effort should go to regions with high mortality is countered by the intuition that within a given area that has been newly allocated, its older half has less live data than its younger half, so it ought to be better to collect the older half.[1] A plot of the survivor function of the distribution of object lifetimes (Figure 1.1) graphically illustrates this fact—the survivor function is always a decreasing function, thus the area under the curve, which is the amount of surviving data, is smaller in the segment $[V/2, V]$ than in the segment $[0, V/2]$. In fact, it would be best to collect just the oldest object in the area. Figure 1.2 illustrates the point: imagine that newly allocated objects are placed in a queue, which plays the role of the youngest generation in generational collectors (also known as the *nursery*). When objects emerge from the queue, they are tested for liveness. Ideally, the length of the queue should be equal to the lifetime of each object, so that no object exits the queue alive. However, objects have diverse lifetimes, which a queue of fixed length cannot match. Yet if objects allocated in close succession have similar lifetimes, then the queue can manage them well, and if it can adjust the queue length to the objects' lifetimes, so much the better. Our design for the deferred older-first collector in Section 3.1.1, p.26, can be seen as an attempt to realize an approximation to a collection queue, within the restrictions of a finite and fixed space available for the queue and its survivors.

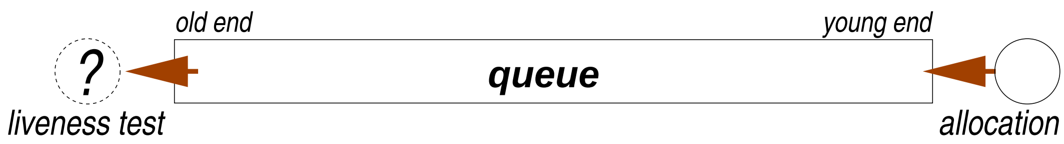

Figure 1.2. A collection queue.

With regard to the empirical observations on which the first tenet was founded, we should point out that, even though a decreasing mortality has been attested in all measured systems

[1]This point is subtle. For instance, in discussing garbage collection in the implementation of SML, Appel states both that *the longer one waits before a collection, the more records have turned into garbage* [Appel, 1992, p. 228], and that *newer cells have a higher proportion of garbage, which is why the garbage collection efforts should be concentrated on them* [Appel, 1992, p. 208]. Both of these statements are true, but not in the same context. If the meaning of characterizations such as these is improperly or simplistically understood, it is easy to derive incorrect inferences.

for relatively young objects, the behavior for relatively older objects, and certainly for very long-lived objects in long-running programs, remains unexplored.

Concerning the second tenet, excess copying: we empirically examined the difference in copying cost between a collector that assumes that everything outside the collected region is live, and one that has an oracle to tell apart the live from the dead. For the collection schemes we considered, we found that, as long as the scheme preserves the objects in a logical order of allocation (thus always having data of a contiguous age group in the region collected), the excess copying is usually not excessive. We then measured the statistics of pointer direction and found that neither older-to-younger nor younger-to-older direction dominates across all studied traces (Smalltalk and Java). But, more importantly, we found that pointer distances (differences in age between the object containing the pointer and the object pointed to) are very small indeed, whether positive or negative.[2] While this property works in favor of generational collection, it also works in favor of any other age-order-preserving collection scheme, by reducing the number of pointers that must be maintained (Section 3.2.2).

Concerning the third tenet, the implementation overheads: it withstood our validation. Indeed, the circumstance that neither pointer direction dominates is not significant, in light of another empirical fact: most pointer stores are into new, or very recently allocated objects. Generational collectors do not need to record these stores, regardless of direction. The other age-based collectors we introduce here must record the younger-to-older pointer stores into new objects, and pay a significant price for it. Nevertheless, we shall see that for many programs, it is worth paying this price to receive the benefit of lower copying cost.

In this study, we explore generational and other collection schemes among those that preserve the relative order of objects, using simulation studies and a prototype collector imple-

[2] DeTreville reports that in Taos, an operating system written in Modula-2+, the younger-to-older pointer stores barely outnumber (by $1.96 \cdot 10^6$ to $1.17 \cdot 10^6$) older-to-younger pointer stores [DeTreville, 1990b, p. 29]. The reported temporal distances are also very short; since they were obtained with respect to a particular garbage collector execution, they are not directly comparable to our measurements of Smalltalk and Java distances in Section 6.2.

mentation. We examine the obvious opposite of youngest-first collection, which is a strict oldest-first: choosing a fixed amount of oldest data. It sometimes works well, but if the chosen amount is small, the collector works poorly. Similarly, a strict youngest-first collector, which always chooses a fixed amount of youngest data (i.e., the nursery), works poorly. These observations lead us to explore strategies in which the collector chooses the collected region more flexibly, and visits both older and younger regions. A deferred older-first collector does so in the simplest possible manner, by moving a window of collection across the heap, but looking at older objects first. It performs remarkably well, sometimes close to the performance of a locally-optimal collector. The intuitive reason for this good performance is also remarkably simple: as the window of collection moves towards younger objects, it finds itself eventually at a position corresponding to an age by which most objects have died; and, since few objects survive in the window, the window does not move very fast. Thus, this collector homes in on the region where it is most profitable to collect, and stays there for a long time. A welcome property is that the best configurations observed in our experiments are those with a small window size, and, since the amount copied per collection is limited by window size, the low total copying cost is accompanied by predictably short pause times. Pointer-maintenance costs are inevitably higher in this collector than they are in generational collection. Nevertheless, the trade-off between these two cost components favors the newly proposed collector for a number of programs we examine.

In Chapter 2, we present the state of scholarship and introduce the terms used in the study of garbage collection algorithms. We discuss a classification of age-based collectors and the design and implementation of new algorithms in Chapter 3. We set the stage for the experimental evaluation of different collectors by examining the intrinsic properties of benchmark programs in Chapter 4. The copying cost of collection is the subject of Chapter 5, while the pointer-maintenance cost is discussed in Chapter 6.

Note on organization. Chapters 4–6 include long series of figures and tables. These are printed *en masse* at the end of each chapter, so as not to disrupt the flow of text, yet keep them close to it.