CHAPTER 2

BACKGROUND

This chapter introduces the concepts and the terminology used in the subsequent discussion while surveying the vast field of literature related to memory management and garbage collection.

2.1 Dynamic memory management

Under the appellation of dynamic memory management, we study the provision in the run-time system of a programming language of mechanisms for allocation of contiguous blocks of memory of arbitrary size and unrestricted temporal extent at the request of the program. These blocks of memory are known as *objects*. In object-oriented languages, there is usually a one-to-one correspondence between a linguistic object and its representation as a dynamically allocated block of memory. In other contexts, the representations of a C `struct` created using a call to `malloc`, a Pascal record created using `new`, or a LISP `cons` cell, are all objects.

A significant amount of research has gone into algorithms for managing dynamically allocated data under the restriction that the objects must not be moved, and that the program must explicitly request object deallocation following its last use of an object. These algorithms concentrate on fast allocation, typically using free lists, and efficient memory usage, avoiding fragmentation. Wilson wrote an excellent survey of the field [Wilson *et al.*, 1995].

Our focus, however, is on algorithms that automatically detect that the last use of an object must have occurred, by proving that no further use of the object is possible. These algorithms are called *garbage collectors*. They remove the burden of object deallocation from the programmer, and in so doing eliminate an important source of programming errors.[1]

The garbage collector considers an object to be garbage (or *dead*) if it is not *reachable*: the program cannot access it by any chain of pointer traversals. Any object that is potentially reachable is considered *live*. Although the name implies that garbage is collected, it is usually the live objects that the collector works with: it preserves them, while reusing the space of the garbage objects.

We shall briefly introduce the concepts needed to understand the arguments made later; for a comprehensive treatment, the reader should consult Cohen's survey of the early development of garbage collection techniques [Cohen, 1981], Wilson's survey, which concentrates on newer algorithms [Wilson, 1992], as well as Jones and Lins's textbook presentation of the field [Jones and Lins, 1996].

2.1.1 Liveness and reachability

Let us first look at simple (non-generational) collection, in which the entire heap is subject to collection. Ideally, any object that will not be used by the program after the time of collection can be discarded as garbage. Unfortunately, this criterion is not effectively computable. Instead, the consensus is to use *pointer reachability* as a conservative approximation. Given a set of root pointers into the heap, owned by the program in its data outside the heap (stack, registers, etc.), everything reachable from this set is considered live. In other words, the transitive closure of the points-to relation, starting with the roots, defines the survivors of the collection. This approximation is safe, since no computation can use an object unless it can refer to it by a pointer, and if it does not already have that pointer it can only obtain it by following some chain of pointers starting from some other pointer it does have. Pointer reachability is simple to compute and is found in the core of all garbage collectors. Between the approximation of

[1] A study of software errors in IBM MVS ascribes almost one half to pointer and array access errors [Sullivan and Chillarege, 1991].

liveness by reachability and the unachievable exact liveness according to future use, there can be a gradation of techniques to make the approximation more accurate using different static analyses of the program (compile-time garbage collection, live variable analysis, etc.). We do not consider these here; hereafter we equate liveness with reachability from the root set of pointers.

We are concerned, however, with another distinction that arises only with region-based collection, which divides the heap H into a collected region C and a remainder U. Live objects in the collected region are those reachable from the set of root pointers. The goal of region-based collection is to avoid having to look at the (normally large) remainder. Instead, a write-barrier ensures that all pointers crossing from U to C are readily available at time of collection. All such pointers, called the *remembered set* for region C, are added to the root set, and then the transitive closure of pointer reachability is computed only within C. Let us call this result S_r, for *survivors by remembered-set reachability*. The actual set of live data is $S \subseteq S_r$. Calculation using remembered-set reachability treats all objects in the uncollected remainder U as if live, since it does not examine U, and thus potentially overestimates the set S. In *any* region-based collector, there is potential for such excess retention or "nepotism" [Ungar and Jackson, 1988] on each garbage collection.

How much excess retention is there? Clearly, a collection scheme could have dramatically high amounts of excess retention if it were the case that (1) pointers tended to cross region boundaries predominantly into the collected region, thus there were many pointers in the remembered-set, *and* (2) many of these pointers were from objects in the remainder that are actually dead, *and* (3) the pointer structure caused the transitive closure operation falsely to mark very large amounts of data as live, starting from such false pointers. The effects on performance would be profound, since both the cost of each collection and the frequency of collections would be high.

We have assumed in the preceding that the set of external roots in the stack and registers is accurately known. This is predicated on the ability of the collector to examine and interpret the stack and registers. A system may opt for self-descriptive data, which incurs the space overhead of tagging each word, or a more compact form using stack maps describing stack contents at admissible garbage collection points. Additionally, a data-flow liveness analysis may reduce the number of registers in the root set [Agesen *et al.*, 1998].

On the contrary, if the system is uncooperative, then it may be the case that neither the values in the stack and registers, nor the values in the heap can be accurately interpreted: it may be impossible to distinguish some pointer representations from some non-pointer representations. Such is the situation with languages such as C and C++, which were designed without garbage collection in mind. *Conservative* garbage collectors (or *ambiguous-roots* collectors) have been developed to cope with this problem [Boehm and Weiser, 1988; Demers *et al.*, 1990; Boehm, 1993]. Although the analyses in this study have been designed and experimentally tested with accurate-roots collection in mind, they may be adapted to an ambiguous-roots setting.

2.1.2 Preserving live data

The collector must preserve the live objects for future use by the program. However, a distinction is made in collector design among techniques that keep objects in place, and those that move them. Mark/sweep collectors do not move objects [McCarthy, 1960]. Instead, free lists are used to manage space, as in manual dynamic storage allocation. The mark phase automates the detection of objects that have become garbage, while the sweep phase invokes the deallocation routine of the underlying dynamic storage allocator for each garbage object in turn. Copying collectors, which are the focus of our study, move the live objects into a new memory area, the "to-space," allowing the entire old area, or "from-space," to be reused for allocation [Minsky, 1963; Fenichel and Yochelson, 1969]. The copying is accompanied by the update of pointers that point to the moved objects. The detection of live objects, by transitive closure for pointer reachability, and the copying into to-space are usually interleaved in a *Cheney scan* [Cheney, 1970] which also eliminates the need for an auxiliary stack to compute the closure. With the use of two *semispaces* allocation is from a contiguous area, hence simple and fast. The collector visits only live objects, which are typically fewer than garbage objects. However, the memory requirement of the copying collector is, in the worst case, double that of the mark/sweep collector.

2.1.3 Incremental collection

Collectors that collect entire heaps at once incur very long pause times. In applications tolerating only short and predictable pauses, incremental collection is needed. The coordination between the collector and the running program to allow the collector to make progress piecemeal, or even concurrently with the program, is not simple. It is accomplished either using a *read-barrier* mechanism, or a *write-barrier* mechanism. With a read barrier, the program is alerted when it tries to follow a pointer to an object that the collector has moved [Baker, 1978]. Because read operations are extremely frequent, read barriers have used hardware support for efficiency [Zorn, 1989].[2] With a write barrier, when the program stores a pointer into an object, the write is recorded. Whereas hardware support for the write barrier can be provided as well, as in the SOAR Smalltalk system [Ungar, 1986], pointer update operations are typically much fewer in number than pointer reads, so a software barrier in the form of in-line code is generally thought to be acceptable.

2.1.4 Generational copying collection

Generational copying collection divides the heap according to the age of objects into multiple areas, or generations, and collects younger generations more frequently than older generations [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984]. There are subtle differences among these algorithms as described in the literature, because most descriptions combine the organization of the heap into generations with particular policies for promotion of survivors of garbage collections. Even when these algorithms are designed as non-incremental ("stop-and-copy"), they can be effectively incremental, by virtue of the fact that most collections are only

---

[2] Zorn finds that a software read barrier could be implemented with a CPU overhead of not more than 20%, however [Zorn, 1990a].

of the youngest generation, and thus most pauses are short. Numerous practical implementations and studies of predictive and adaptive management of generations have reported good performance [Caudill and Wirfs-Brock, 1986; Courts, 1988; Shaw, 1988; Sobalvarro, 1988; Ungar and Jackson, 1988; Wilson, 1989; Wilson and Moher, 1989; Appel, 1989b; Wilson *et al.*, 1991; Hudson *et al.*, 1991; Stefanović, 1993b; Stefanović and Moss, 1994; Diwan *et al.*, 1995; Barrett and Zorn, 1995].

The performance of a collector is affected by a number of factors: the number of generations in the system; the promotion policy, or how long an object must remain in one generation before it is advanced into the next older; when to initiate collection; and how to track the pointers from older to younger generations [Wilson, 1992]. This space of design choices is immense. Moreover, the allocation characteristics of languages and individual programs are varied, and they strongly affect the performance of collectors. The performance metrics—time overhead, pause time, and space overhead—are insidiously hard to define and measure. Does the time overhead include the write barrier, and how can the cost of the write barrier be extricated from other run-time costs? Is maximum pause time more important than average pause time, or some percentile of the distribution thereof? Should one measure the peak memory usage or the average memory usage? Should peak usage reflect the space needed only during copying to hold the to-space? Is it reasonable to let the oldest generation be "tenured" (allowed to grow indefinitely and never collected) or should measurements be taken with respect to a constant available space? The answers to these questions must depend on the intended application. As a result of the inherent complexity of generational collector design and of the methodological difficulties in evaluation, the understanding of generational garbage collectors has remained incomplete.

Aside from remarks in Wilson's survey [Wilson, 1992] and Jones and Lins' book [Jones and Lins, 1996, p.151], which admit the possibility of collecting an older generation independently of a younger one, generational collectors are youngest-first: when collecting an older generation, all younger ones must also be collected. The number of generations is typically small, so the choice of the size of the collected region is limited. Barrett and Zorn examined an adaptive scheme which dynamically chose an age boundary for each collection [Barrett and Zorn, 1995]. Our work instead proposes to collect among older (but not necessarily oldest) objects preferentially, postponing the youngest objects until they have had time to become older and die. The work closest to ours in this respect is Clinger and Hansen's "non-predictive" collector, which organizes objects according to the time elapsed since last collected and performs well with a hypothetical exponential distribution of lifetimes [Clinger and Hansen, 1997]. However, as we shall see in our empirical study in Chapter 5, that particular organization does not work remarkably well.

2.1.4.1 Pointer maintenance

In order to collect one region (a set of youngest generations) without collecting the remainder of the heap, the set of pointers crossing the boundary from the remainder and into the collected region must be known. Pointers installed in an object at the time of its creation can only point to older objects and can never cross the boundary in the interesting direction, only pointers installed by *updates* can. Each updating pointer store must be checked: if the stored pointer is $p \to q$, and p is in an older generation than q, then a record of this store will allow the pointer to be found at the time of collection of q. Early generational collectors used indirect pointers and region entry and exit tables, mechanisms that are very expensive without hardware support for pointer operations, and not transparent to the language. Ungar's collector used a *remembered set* of updated objects in the old (tenured) generation [Ungar, 1984]. A set can alternatively record the precise location of the updated object fields ("slots"), trading extra space for reduced time to find the updated fields at garbage collection time [Hosking *et al.*, 1992]. If there are few repeated stores into the same location between collections, a data structure simpler than a set may be preferable, such as a store list [Appel, 1989b] or a sequential store buffer [Hosking *et al.*, 1992].

Virtual memory mechanisms can be used to record the approximate location of updated object fields to within a page. Pages are write-protected, the first update of a page causes a page protection fault, and the software fault handler makes a note of the page (in an auxiliary bit array) and unprotects it. This scheme is called *page marking*. Unfortunately, virtual memory primitives in current operating systems are not well adapted to the task (non-pointer stores and null pointer stores equally cause protection faults) and are slow (because of user/kernel mode switching) [Zorn, 1990a]; moreover, the granularity of pages is too coarse (the ratio of page size to the typical number of pointer updates within it is high) [Hosking *et al.*, 1992]. Software surrogates of page marking are known as *word marking* [Sobalvarro, 1988], or more generally, *card marking*, in which the address space is conceptually divided into sub-page units (cards). Each pointer store is augmented with code to index into an auxiliary card table and set the bit corresponding to the card of the updated field. Whereas word marking records the precise location of updated fields, card marking with cards larger than a word saves auxiliary table space at the cost of scanning individual cards at garbage collection time [Hosking *et al.*, 1992]. Card tables as described tend to retain scattered marked entries over time, and it is better to summarize them at collection time (into a concise remembered-set representation) so that the card table reflects only incremental updates [Hosking and Hudson, 1993].

2.1.5 Mature spaces

Bishop [Bishop, 1977] and later Hudson and Moss [Hudson and Moss, 1992] proposed dividing the heap into spaces managed by different algorithms. Bishop allows several independent areas, not necessarily all garbage-collected, and uses lists of incoming and outgoing inter-area links. Hudson and Moss divide the heap by age into a young and a mature space, which is in agreement with the intuition that the patterns of pointer structure linking the objects lose the strong directional bias (see Section 6.2) after a certain age, hence the generational algorithms, which exploit this bias, lose efficiency. The mature space is divided into areas of bounded size (or blocks of fixed size in an implementation in the Beta language [Garup and

Seligmann, 1993]), which are processed one at a time. It is assumed that a generational algorithm manages the young space and engages the incremental mature space algorithm only when promoting out of the oldest generation of the young space.

2.2 Object survival and mortality

A recent book on garbage collection states [Jones and Lins, 1996, p.144]:

> *The insight behind generational garbage collection is that storage reclamation can be made more efficient and less obtrusive by concentrating effort on reclaiming those objects most likely to be garbage, i.e., young objects.*

Hayes and others observe that youngest objects have the highest *mortality* [Hayes, 1991; Baker, 1993; Hayes, 1993; Stefanović and Moss, 1994], which is a refinement of the somewhat vague observation that most objects have a very short lifetime while some live much longer [Lieberman and Hewitt, 1983; Ungar, 1984; Shaw, 1988; DeTreville, 1990a; Zorn, 1990b]. Figure 2.1 graphs a typical mortality function using Tomcatv written in Smalltalk as an example (see Section 4.2.2.3), with the age of objects on the horizontal axis, and for a given age, the rate at which they die on the vertical axis. Both axes use a logarithmic scale.

Is high mortality the same as being most likely to be garbage? No, since the very newest objects are patently not garbage. In statistical terms, it is the *survivor* function, and not the mortality, that tells us what is live, and the survivor function is always 1 at age 0. Formally, the survivor function $s(t)$ is the fraction of original allocation that is still live at age t. The mortality $m(t) = -\frac{s'(t)}{s(t)}$ is the age-specific death rate, or likelihood that an object will die in the next instant provided it has survived until age t [Cox and Oakes, 1984; Elandt-Johnson and Johnson, 1980; Baker, 1993]. Figure 2.2 plots a typical survivor function, again using Tomcatv. The horizontal axis is object age on a logarithmic scale. The vertical axis is the fraction of original allocation still live at a given age.

Given a region of just-allocated data of size V, is it more profitable to collect its younger half, ages 0 to $V/2$, or its older half, ages $V/2$ through V? Statistically speaking, the answer

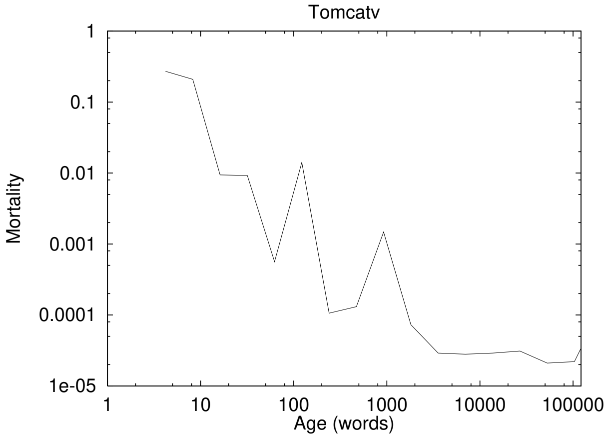

Figure 2.1. A typical object mortality function.

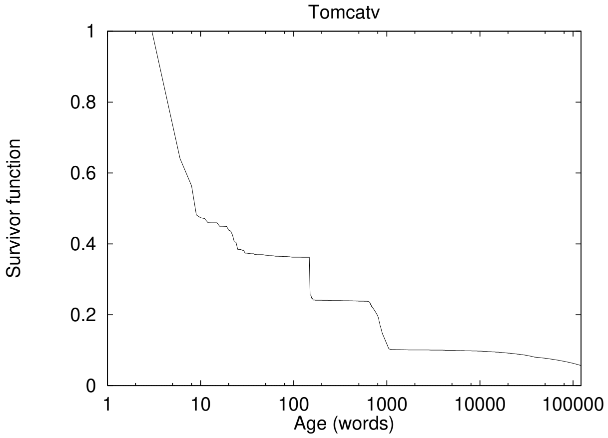

Figure 2.2. A typical object survivor function.

must be to collect the older half, which is clear from the survivor function diagram, Figure 1.1, or the simple calculation: $\int_{V/2}^{V} s(t)dt \leqslant \int_{0}^{V/2} s(t)dt$, because the survivor function $s(t)$ is non-increasing.

In practice, we have to decide where to put the survivors, which complicates this simple analysis. The analysis suggests, however, the following revision of the preceding quotation:

> *Generational collection is efficient, provided its generations are well configured, because it concentrates on a set of young objects that includes the objects most likely to be garbage, albeit along with objects least likely to be garbage.*

If the collector can concentrate *only* on the objects most likely to be garbage, then it may be able to achieve even lower copying cost. This discussion was entirely independent of the actual shape of the survivor and mortality curves, but the relative merits of generational and other collectors will certainly depend on them.

Assumed or measured object lifetime distributions can be used to predict the performance of collectors [Clinger and Hansen, 1997]. However, the implicit assumption that object lifetimes are independent identically distributed random variables is far from true: phase behavior of programs must lead to highly correlated object lifetimes. The collector performance calculations are tied to the assumption of a steady state, but this assumption restricts the class of admissible distributions. The questions of the feasibility and the utility of analytical modelling of object lifetimes therefore remain open [Stefanović *et al.*, 1998a; Appel, 1997; Stefanović *et al.*, 1998b].

2.3 Theoretical models of memory management

Object-based computation appears in its purest form in the Storage Modification Machine, or pointer machine [van Emde Boas, 1990; Kolmogorov and Uspenskii, 1958; Schönhage, 1980], a model of computation based on a finite directed graph structure of storage instead of an array of memory registers as in the more commonly used RAM model. The graph structure is dynamic. The model incorporates the notion of new node creation. The requirement for what we call garbage collection is expressed abstractly in the following dictum [van Emde Boas, 1990, p. 32]: "The map p^* does not have to be surjective; however, nodes which can not be reached by tracing a word w in Δ^* starting from the center a will play no subsequent role during the computations of the SMM, and therefore nodes may be assumed to have disappeared when they become unreachable."

A complementary aspect of theoretical approaches is the development of semantic models of memory management: deriving the properties of dynamically allocated data by linguistic means, usually in the context of typed functional languages [Morrisett *et al.*, 1995]. These models indicate which objects are garbage and why, and additionally allow one to express memory management actions syntactically, and then reason about their correctness. (Unfortunately, these models do not answer quantitative questions of the kind posed in our study.) Similar models are desirable for memory management in object-oriented languages, but their semantic modelling in general remains less settled [Abadi and Cardelli, 1996]. A related discipline is that of compile-time garbage collection, or type and data-flow analyses to improve the run-time performance of collection in (usually) functional languages [Appel, 1989a; Harrison, 1989; Baker, 1990; Goldberg and Gloger, 1992; Boehm and Shao, 1993; Fradet, 1994; Aditya *et al.*, 1994; Mohnen, 1995]. The primary idea is to eliminate the need for heap allocation in the first place, by proving that certain objects can be allocated on the stack instead, which is considered (although not universally, cf. [Appel, 1987]) to be faster. More ambitious analyses aim to provide hints to a generational collector about object lifetimes [Yi and Harrison, 1992].