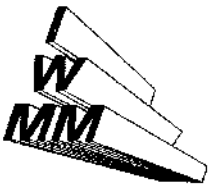
Y. Bekkers J. Cohen (Eds.)

Memory Management

International Workshop IWMM 92
St. Malo, France, September 17-19, 1992
Proceedings



Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Volume Editors

Yves Bekkers
IRISA, Campus de Beaulieu
F-35042 Rennes, France

Jacques Cohen
Mitchum School of Computer Science, Ford Hall
Brandeis University, Waltham, MA 02254, USA

CR Subject Classification (1991): D.1, D.3-4, B.3, E.2

ISBN 3-540-55940-X Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55940-X Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author/editor
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

Storage reclamation became a necessity when the Lisp function *cons* was originally conceived[1]. That statement is simply a computer-oriented version of the broader precept: Recycling becomes unavoidable when usable resources are depleted. Both statements succinctly explain the nature of the topics discussed in the *International Workshop on Memory Management* (IWMM) that took place in Saint-Malo, France, in September 1992. This volume assembles the refereed technical papers which were presented during the workshop.

The earlier programming languages (such as Fortran) were designed so that the size of the storage required for the execution of a program was known at compile time. Subsequent languages (such as Algol 60) were implemented using a *stack* as a principal data-structure which is managed dynamically: information pushed onto a stack uses memory space which can be later released by popping.

With the introduction of structures (also called records) in more recent programming languages, it became important to establish an additional run-time data structure: the *heap*, which is used to store data-cells containing pointers to other cells. The *stack-heap* arrangement has become practically universal in the implementation of programming languages. An important characteristic of the cells in the heap is that the data they contain can become "useless" since they are not pointed to by any other cells. Reclamation of the so-called "useless cells" can be performed in an *ad hoc* (manual) manner by having the programmer explicitly return those cells to the run-time system so that they can be reused. (In *ad hoc* reclamation the programmer has to exercise great caution not to return cells containing valuable data.) This is the case of languages like Pascal or C which provide primitive procedures for returning useless cells. In the case of languages such as Lisp and Prolog reclamation is done automatically using a run-time process called *garbage-collection* which detects useless cells and makes them available for future usage.

Practically all the papers in this volume deal with the various aspects of managing and reclaiming memory storage when using a *stack-heap* model. A peculiar problem of memory management strategies is the unpredictability of computations. The undecidability of the halting problem implies that, in general, it is impossible to foresee how many cells will be needed in performing complex computations.

There are basically two approaches for performing storage reclamation: one is *incremental*, i.e., the implementor chooses to blend the task of collecting with that of actual computation; the other is what we like to call the *mañana* method - wait until the entire memory is exhausted to trigger the time-consuming operation of recognizing useless cells and making them available for future usage. A correct reclamation should ensure the following properties:

- *No used cell will be (erroneously) reclaimed.*
- *All useless cells will be reclaimed.*

Violating the first property is bound to have tragic consequences. A violation of the second may not be disastrous, but could lead to a premature halting of the execution due to the lack of memory. As a matter of fact, *conservative* collectors have been proposed to trade a (small) percentage of unreclaimed useless cells for a speedup of the collection process.

An important step in the collection is the identification of useless cells. This can be achieved by *marking* all the useful cells and *sweeping* the entire memory to collect useless

[1] The reader is referred to the chapter on the *History of Lisp*, by John McCarthy, which appeared in History of Programming Languages, edited by Richard L. Wexelblat, Academic Press, 1981, pp 173-183.

(unmarked) cells. This process is known as *mark-and-sweep*. Another manner of identifying useless cells is to keep *reference counts* which are constantly updated to indicate the number of pointers to a given cell. When this number becomes zero the cell is identified as useless. If the mark-and-sweep or the reference count techniques fail to locate any useless cells, the program being executed has to halt due to lack of storage. (A nasty situation may occur when successive collections succeed in reclaiming only a few cells. In such cases very little actual computation is performed between consecutive time-consuming collections.)

Compacting collectors are those which compact the useful information into a contiguous storage area. Such compacting requires that pointers be properly readjusted. Compacting becomes an important issue in paging systems (or in the case of hierarchical or virtual memories) since the compacted useful information is likely to result in fewer page faults, and therefore in increased performance.

An alternative method of garbage-collection which has drawn the attention of implementors in recent years is that of *copying*. In this case the useful cells are simply copied into a "new" area from the "old" one. These areas are called semi-spaces. When the space in the "new" area is exhausted, the "old" and "new" semi-spaces are swapped. Although this method requires twice the storage area needed by other methods, it can be performed incrementally, thus offering the possibility of *real-time* garbage-collection, in which the interruptions for collections are reasonably short.

The so-called *generational* garbage-collection is based on the experimental fact that certain cells remain used during substantial periods of the execution of a program, whereas others become useless shortly after they are generated. In these cases the reclaiming strategy consists of bypassing the costly redundant identification of "old generation" cells.

With the advent of *distributed* and *parallel* computers reclamation becomes considerably more complex. The choice of storage management strategy is, of course, dependent on the various types of existing architectures. One should distinguish the cases of:

1. Distributed computers communicating via a network,
2. Parallel shared-memory (MIMD) computers, and
3. Massively parallel (SIMD) computers.

In the case of distributed reclamation it is important that collectors be fault tolerant: a failure of one or more processors should not result in loss of information. The term *on-the-fly* garbage-collection is (usually) applicable to parallel shared-memory machines in which one or more processors are dedicated exclusively to collecting while others, called *mutators*, are responsible for performing useful computations which in turn may generate useless cells that have to be reclaimed.

Some features of storage management are *language-dependent*. Presently, one can distinguish three major paradigms in programming language design: *functional*, *logic*, and *object-oriented*. Although functional languages, like Lisp, were the first to incorporate garbage-collection in their design, both logic and object-oriented language implementors followed suit. Certain languages have features that enable their implementors to take advantage of known properties of data in the stack or in the heap so as to reduce the execution time needed for collection and/or to reclaim as many useless cells as possible.

In the preceding paragraphs we have briefly defined the terms: *mark-and-sweep*, *reference count*, *compacting*, *copying*, *incremental*, *generational*, *conservative*, *distributed*, *parallel*, *on-the-fly*, *real-time*, and *language-dependent features*. These terms should serve to guide the reader through the various papers presented in this volume.

We suggest that non-specialists start by reading the three survey papers. The first provides a general overview of the recent developments in the field; the second specializes in distributed collection, and the third deals with storage management in processors for logic programs. The other chapters in this volume deal with the topics of distributed, parallel, and

incremental collections, collecting in functional, logic, and object-oriented languages, and collections using massively parallel computers. The final article in this volume is an invited paper by H. G. Baker in which he proposes a "reversible" Lisp-like language (i.e., capable of reversing computations) and discusses the problems of designing suitable garbage-collectors for that language.

We wish to thank the referees for their careful evaluation of the submitted papers, and for the suggestions they provided to the authors for improving the quality of the presentation. Finally, it is fair to state that, even with technological advances, there will always be limited memory resources, especially those of very fast access. These memories will likely remain costlier than those with slower access. Therefore many of the solutions proposed at the IWMM are likely to remain valid for years to come.

July 1992

Yves Bekkers
Jacques Cohen

Program Committee

Chair	
Jacques Cohen	Brandeis University, Waltham, MA, USA
Members	
Joel F. Bartlett	DEC, Palo Alto, CA, USA
Yves Bekkers	INRIA-IRISA, Rennes, France
Hans-Jurgen Boehm	Xerox Corporation, Palo Alto, CA, USA
Maurice Bruynooghe	Katholieke Universiteit, Leuven, Belgium
Bernard Lang	INRIA, Le Chesnay, France
David A. Moon	Apple Computer, Cambridge, MA, USA
Christian Queinnec	Ecole Polytechnique, Palaiseau, France
Dan Sahlin	SICS, Kista, Sweden
Taiichi Yuasa	Toyohashi Univ. of Tech., Toyohashi, Japan

We thank all the people who helped the program committee in the refereering process, some of whom are listed below: K. Ali, M. Banâtre, P. Brand, A. Callebou, P. Fradet, S. Jansson, P. Magnusson, A. Mariën, R. Moolenaar, A. Mulkers, O. Ridoux, A. Saulsbury, T. Sjöland, L. Ungaro, P. Weemeeuw.

Workshop Coordinator

Yves Bekkers — INRIA-IRISA, Rennes, France

Sponsored by

INRIA
University of Rennes I
CNRS-GRECO Programmation

In cooperation with

ACM SIGPLAN

Table of Contents

Logic Programming Languages I

Object Oriented Languages

Logic Programming Languages II

Incremental

Improving Locality

Parallelism II

Massively Parallel Architectures

Invited Speaker

Uniprocessor Garbage Collection Techniques

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find data objects[1] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling pointer" into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

[1] We use the term object loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.

Since liveness is a *global* property, this introduces nonlocal bookkeeping into routines that might otherwise be orthogonal, composable, and reusable. This bookkeeping can reduce extensibility, because when new functionality is implemented, the bookkeeping code must be updated.

The unnecessary complications created by explicit storage allocation are especially troublesome because programming mistakes often introduce erroneous behavior that breaks the basic abstractions of the programming language, making errors hard to diagnose.

Failing to reclaim memory at the proper point may lead to slow memory *leaks*, with unreclaimed memory gradually accumulating until the process terminates or swap space is exhausted. Reclaiming memory too soon can lead to very strange behavior, because an object's space may be reused to store a completely different object while an old pointer still exists. The same memory may therefore be interpreted as two different objects simultaneously with updates to one causing unpredictable mutations of the other.

These bugs are particularly dangerous because they often fail to show up repeatably, making debugging very difficult; they may never show up at all until the program is stressed in an unusual way. If the allocator happens not to reuse a particular object's space, a dangling pointer may not cause a problem. Later, in the field, the application may crash when it makes a different set of memory demands, or is linked with a different allocation routine. A slow leak may not be noticeable while a program is being used in normal ways—perhaps for many years—because the program terminates before too much extra space is used. But if the code is incorporated into a long-running server program, the server will eventually exhaust its swap space, and crash.

Explicit allocation and reclamation lead to program errors in more subtle ways as well. It is common for programmers to statically allocate a moderate number of objects, so that it is unnecessary to allocate them on the heap and decide when and where to reclaim them. This leads to fixed limitations on software, making them fail when those limitations are exceeded, possibly years later when memories (and data sets) are much larger. This "brittleness" makes code much less reusable, because the undocumented limits cause it to fail, even if it's being used in a way consistent with its abstractions. (For example, many compilers fail when faced with automatically-generated programs that violate assumptions about "normal" programming practices.)

These problems lead many applications programmers to implement some form of application-specific garbage collection within a large software system, to avoid most of the headaches of explicit storage management. Many large programs have their own data types that implement reference counting, for example. Unfortunately, these collectors are often both incomplete and buggy, because they are coded up for a one-shot application. The garbage collectors themselves are therefore often unreliable, as well as being hard to use because they are not integrated into the programming language. The fact that such kludges exist despite these problems is a testimony to the value of garbage collection, and it suggests that garbage collection should be part of programming language implementations.

In the rest of this paper, we focus on garbage collectors that are built into a language implementation. The usual arrangement is that the allocation routines of

the language (or imported from a library) perform special actions to reclaim space, as necessary, when a memory request is not easily satisfied. (That is, calls to the "deallocator" are unnecessary because they are implicit in calls to the allocator.)

Most collectors require some cooperation from the compiler (or interpreter), as well: object formats must be recognizable by the garbage collector, and certain invariants must be preserved by the running code. Depending on the details of the garbage collector, this may require slight changes to the code generator, to emit certain extra information at compile time, and perhaps execute different instruction sequences at run time. (Contrary to widespread misconceptions, there is no conflict between using a compiled language and garbage collection; state-of-the art implementations of garbage-collected languages use sophisticated optimizing compilers.)

1.2 The Two-Phase Abstraction

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again. Such data objects are referred to as *garbage*. The basic functioning of a garbage collector consists, abstractly speaking, of two parts:

1. Distinguishing the live objects from the garbage in some way, or *garbage detection*, and
2. Reclaiming the garbage objects' storage, so that the running program can use it.

In practice, these two phases may be functionally or temporally interleaved, and the reclamation technique is strongly dependent on the garbage detection technique.

In general, garbage collectors use a "liveness" criterion that is somewhat more conservative than those used by other systems. In an optimizing compiler, a value may be considered dead at the point that it can never be used again by the running program, as determined by control flow and data flow analysis. A garbage collector typically uses a simpler, less dynamic criterion, defined in terms of a *root set* and *reachability* from these roots. At the point when garbage collection occurs[2] all globally visible variables of active procedures are considered live, and so are the local variables of any active procedures. The *root set* therefore consists of the global variables, local variables in the activation stack, and any registers used by active procedures. Heap objects directly reachable from any of these variables could be accessed by the running program, so they must be preserved. In addition, since the program might traverse pointers from those objects to reach other objects, any object reachable from a live object is also live. Thus the set of live objects is simply the set of objects on any directed path of pointers from the roots.

Any object that is not reachable from the root set is garbage, i.e., useless, because there is no legal sequence of program actions that would allow the program to reach that object. Garbage objects therefore can't affect the future course of the computation, and their space may be safely reclaimed.

[2] Typically, this happens when allocation of an object has been attempted by the running program, but there is not sufficient free memory to satisfy the request. The allocation routine calls a garbage collection routine to free up space, then allocates the requested object.

1.3 Object Representations

Throughout this paper, we make the simplifying assumption that heap objects are self-identifying, i.e., that it is easy to determine the type of an object at run time. Implementations of statically-typed garbage collected languages typically have hidden "header" fields on heap objects, i.e., an extra field containing type information, which can be used to decode the format of the object itself. (This is especially useful for finding pointers to other objects.)

Dynamically-typed languages such as Lisp and Smalltalk usually use *tagged* pointers; a slightly shortened representation of the hardware address is used, with a small type-identifying field in place of the missing address bits. This also allows short immutable objects (in particular, small integers) to be represented as unique bit patterns stored directly in the "address" part of the field, rather than actually referred to by an address. This tagged representation supports polymorphic fields which may contain either one of these "immediate" objects or a pointer to an object on the heap. Usually, these short tags are augmented by additional information in heap-allocated objects' headers.

For a purely statically-typed language, no per-object runtime type information is actually necessary, except the types of the root set variables.[3] Once those are known, the types of their referents are known, and their fields can be decoded [App89a, Gol91]. This process continues transitively, allowing types to be determined at every pointer traversal. Despite this, headers are often used for statically-typed languages, because it simplifies implementations at little cost. (Conventional (explicit) heap management systems often use object headers for similar reasons.)

2 Basic Garbage Collection Techniques

Given the basic two-part operation of a garbage collector, many variations are possible. The first part, distinguishing live objects from garbage, may be done in several ways: by *reference counting*, *marking*, or *copying*. [4] Because each scheme has a major influence on the second part (reclamation) and on reuse techniques, we will introduce reclamation methods as we go.

2.1 Reference Counting

In a reference counting system [Col60], each object has an associated count of the references (pointers) to it. Each time a reference to the object is created, e.g., when a pointer is copied from one place to another by an assignment, the object's count is incremented. When an existing reference to an object is eliminated, the count is

[3] *Conservative* garbage collectors [BW88, Wen90, BDS91, WH91] are usable with little or no cooperation from the compiler—not even the types of named variables—but we will not discuss them here.

[4] Some authors use the term "garbage collection" in a narrower sense, which excludes reference counting and/or copy collection systems; we prefer the more inclusive sense because of its popular usage and because it's less awkward than "automatic storage reclamation."