Incremental Incrementally Compacting Garbage Collection

Bernard LANG and Francis DUPONT

INRIA, Rocquencourt
B.P. 105, Le Chesnay
78153 France
{*lang,dupont*}*@inria.inria.fr*[1]

Abstract: A mixed-strategy garbage collection algorithm is presented, which combines mark-and-sweep and copy collection. The intent is to benefit from the compacting and linearizing properties of copy collection without losing computational use of half the memory. The stop-and-collect version of the algorithm is a simple and cheap technique to fight memory fragmentation. The collection strategy may be dynamically adapted to minimize the cost of collection, according to the amount of memory actually accessed by the computing process. The parallel version of the algorithm is to our knowledge the only parallel compacting collector for varisized cells, that leaves most (more than half) of the memory available for the computing process.

1 Introduction

The purpose of techniques known as *"garbage collection"* or *scavenging* is to provide automatic reclamation of memory cells that have become useless to a running program because it can no longer access them. Two families of garbage collection algorithms have been developed. The first one is based on an on-the-fly *"reference count"* scheme [Col 60, DeuB 76] that keeps track of the number of pointers accessing a given cell. The second family identifies useful cells by *tracing* all accessible cells from a collection of root accesses which are essentially the named variables of the program being executed. An extensive survey of both families with a discussion of their merits and demerits may be found in [Coh 81].

The algorithms we propose here belong to the second family. This family itself contains two types of algorithms usually called *mark-and-sweep* collection [McC 60] and *copy* collection [Min 63, FenY 69]. Originally these algorithms were developed to work in a *stop-and-collect* mode, i.e. the user program has to stop for some time (typically several seconds) while collection is taking place. Later *non-stop* versions of both types of algorithms were developed (see [Ste 75, Wad 76, KunS 77, DijLMSS 78, Yua 86] for mark-and-sweep and [Bak 78, Daw 82, Bro 84, Hal 84, BekCRU 86] for copy collection). The non-stop techniques include incremental, parallel (multiprocessor) or real-time collection. They are essentially similar and the distinction will be overlooked most of the time in this paper. In practice specific aspects have to be carefully worked out, especially to avoid explicit synchronization for parallel algorithms, or to ensure real-time behavior by locking the pace of the collector and the *mutator* (i.e. the user's program, following the terminology of [DijLMSS 78]).

The issues we wish to address concern single space garbage collection and compaction, either in stop-and-collect or in non-stop mode. We briefly mention relations with generation collectors [LieH 83, Moo 84, Ung 84], but we ignore problems related to distributed garbage collection [Rud 86].

2 A unified view of garbage collection algorithms

In this section we show how all tracing algorithms (i.e. the second family mentioned in the introduction) may be derived from a unique abstract description. The description of the new algorithm proposed in this paper will then be based on the structural uniformity of existing algorithms.

[1] Current address (2003): `Bernard.Lang@inria.fr` and `Francis.Dupont@enst-bretagne.fr`
See end of paper for history and copyright.

2.1 The abstract garbage collection algorithm

The purpose of a tracing collection algorithm is to identify all memory cells that are **not** accessible by the mutator (the user's program) at some point in its execution, so as to make them available for the satisfaction of future memory allocation requests. An *accessible* cell is a cell that may be reached through some linked chain of cells from a collection of *"root pointers"* which are approximately the mutator's pointer variables, either global or allocated on the execution stack of the mutator. It is assumed that there is some available function to identify all the root pointers, and that one can also determine all the pointers contained in an accessible cell.

The abstract collection algorithm is based on the consideration of three sets of memory cells (see figure 1):

- the *universal* set, i.e. the set of all available memory cells, accessible or not;
- the *visited* set, i.e. the set of cells already visited by the collector and thus known to be accessible (or having been such at some point during the collection cycle in the case of non-stop collectors); its complement with respect to the universal set is called the *unvisited* set;
- the *untraced* set, which is a subset of the visited set (more precisely the untraced set should be a set of pointers occurring in cells of the visited set). It is the set of visited cells pointing to cells that may not have been visited yet. Its complement with respect to the visited set is called the *traced* set. An invariant of the algorithm is that there is never a pointer from a traced cell to an unvisited cell. At the beginning of collection, all cells (accessible or not) are unvisited. A scavenging algorithm transfers all accessible cells from the unvisited set into the visited set, and leaves all inaccessible cells (i.e. garbage) in the unvisited set. In the end, all cells left in the unvisited set may then be collected for reuse.

The scavenging algorithm proceeds by first transferring from the unvisited set to the untraced and visited sets all cells that are directly pointed at by a root pointer. Then, as long as there are cells left in the untraced set, these cells are processed as follows: for each untraced cell C, we transfer all cells directly pointed at by some pointer stored in that cell C from the unvisited set to the untraced part of the visited set, unless they are already in the visited set; then the cell C itself is removed from the untraced set, i.e. is moved to the traced set (see figure 2).

The process terminates when the untraced set is empty. This ultimately happens since the number of cells is finite and cells are entered at most once in the untraced set.

Efficiency may be improved by tracing separately (i.e. at different times) the pointers contained in a cell. This may allow better linearization of cells [Arn 72, Cla 76, Daw 82], or better cooperation with the virtual memory [Moo 84]. Thus it would be more precise to consider sets of pointers, but the point is not really essential here.

This abstract view is not really original. The sets considered here correspond to (some combination of) the cell colors used in the description of some collection algorithms [Kung 76, DijLMSS 78, Hal 84, HicC 84]. However, our view is more abstract in the sense that we consider logical rather than physical cells in the scavenging process. Thus when actually implementing the scavenging algorithm, we allow any known programming technique (within efficiency constraints) to indicate that a cell belongs to one set or another. In the case of copy collection, a logical cell may actually be represented by two physical cells. Abstract similarity between different garbage collectors was previously suggested by some authors (e.g. [Coh 81 (p.351)]). In [Arn 72], Arnborg actually evolves a mark-and-sweep collector into a copy collector.

Cells left over in the unvisited set may or may not be located in contiguous chunks of memory. In the latter case two courses of action are possible:

- compaction of the memory by relocation of accessible cells at one end and inaccessible ones at the other end [HadW 67, Weg 72a, LanW 72, Ste 75, FitN 78, Mor 78, Jon 79, CohN 83]
- hole management: all garbage chunks of memory are linked in a *free-list* from which allocation may be performed according to a variety of strategies [Knu 68, Sho 75, KorK 85].

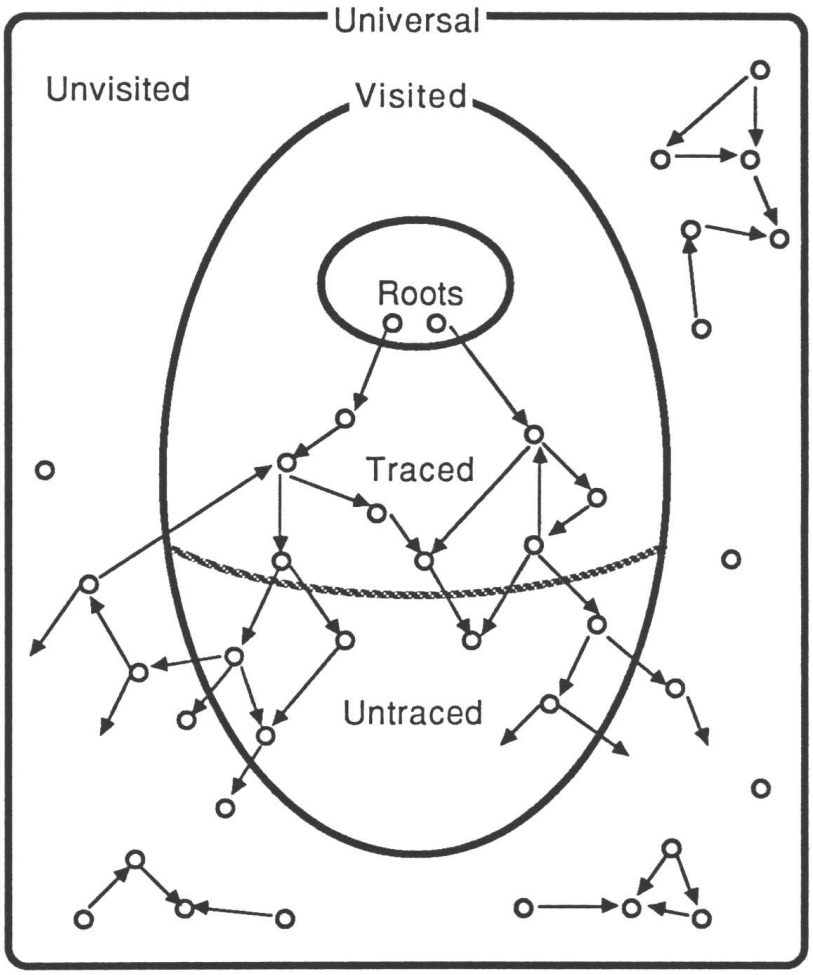

Figure 1: Abstract structure of a tracing collector

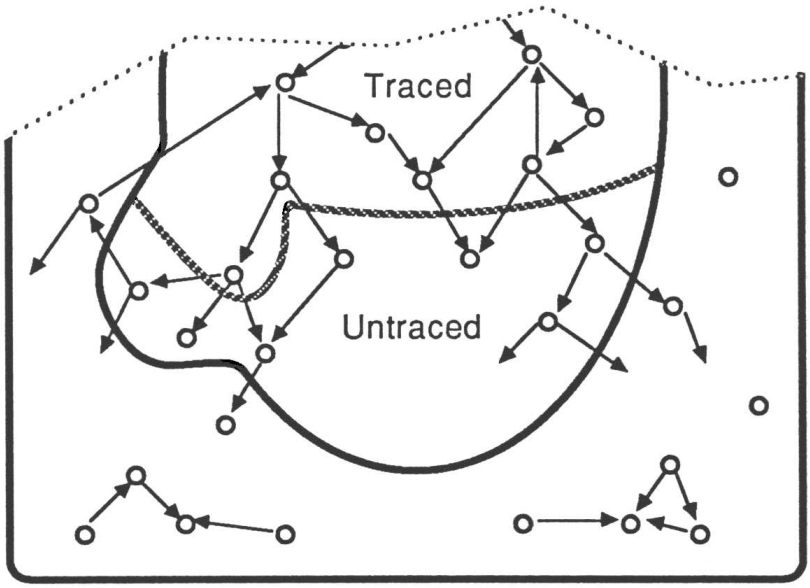

Figure 2: Tracing a cell in the untraced set

2.2 The mark-and-sweep implementation

Several techniques are commonly used to indicate that a cell belongs to a set, among which:

1. use of a special tag bit on each cell which is on iff the cell is in the set considered; the tag bit may be physically separated from the cell;
2. physically locating cells in the same memory area when they belong to the set considered;
3. keeping in some storage structure a list of references to the cells in the set;
4. testing a predicate on the contents of the cell;
5. some combination of the above.

The mark-and-sweep collection algorithms are based on the first technique for the visited set (thus often called the *marked* set). The untraced set is then usually implemented with some variation of the third technique, i.e. pointers to untraced cells are kept in a storage structure which is often a pushdown stack. However [DijLMSS 78] implicitly uses tag bits for the untraced set to emphasize simplicity over efficiency (though with the remark that the third above technique is better).

In [Hib 80, Ben 84] the untraced set is represented more computationally as composed of all marked cells that contain pointers to unmarked ones. This is essentially equivalent to [DijLMSS 78] solution, up to the cost of checking the marking of pointed cells. These last three proposals require repeated and costly sweeps of memory in search for untraced cells. The overhead is limited in the case of [Hib 80] by the additional use of a storage structure for untraced cells (actually a recursion stack), which is unfortunately private to the collector.

Thus, a major defect of the tag bit technique is that its use requires (sometimes repeatedly) sweeping the whole memory, independently of the actual size of the set of cells it identifies. Representing a set as a collection of pointers in a storage structure may be storage costly when this set is large. Since the untraced set could be as large as (half) the number of accessible cells (in the worst case for binary cells), this issue has led to a variety of proposals to improve representation of the untraced set (e.g. [SchW 67, Knu 68 (pp. 417-418), Weg 72b]).

2.3 The copy implementation

Copy collection algorithms are based on the second of the above techniques for set membership implementation. At the beginning of a collection cycle, all accessible cells are assumed to be located in a memory area called *from-space*. During collection, all visited cells are moved in a new memory area called *to-space*, and distinction between visited and unvisited cells is based on their addresses. An invisible forwarding pointer to its new location is kept in the old location of each visited cell, in order to maintain the logical organization of cells. The untraced set is also identified geographically as (all pointers to from-space in) a sub-area of the memory area used for the visited set as in [Che 70, Daw 82, Moo 84], or by some more complex storage structure [Cla 76].

The main drawback of copy scavengers is that they require that only half the available memory space be effectively usable by the mutator. The other half is used to implement the visited set into which accessible cells are transferred by the copying process.

On the other hand, copy scavengers present an essential advantage: they can *linearize* the organization of cells in the memory, i.e. they can copy in contiguous locations cells that directly point to each other. Since cells usually contain several pointers to other cells (typically 2 in Lisp), this linearization is usually deliberately biased towards contiguity of cells related by the most frequently used pointers (*cdr* in the case of Lisp, or according to user supplied information in other systems [Arn 72]).

Linearization improves the locality of data in memory and thus may considerably improve performances in virtual memory systems [BobM 67]. In the case of Lisp (mainly), linearization improves the effectiveness of *"list compression"* techniques known as *cdr-coding* [Han 69, Che 70, BobC 79, LiH 86]. They essentially amount to the elimination of the cdr-pointer of a cell when the cell to which it points follows immediately in memory, at the cost of two extra bits per cell.

2.4 Non-stop implementation modes

The original implementations of mark-and-sweep and copy collection were realized in a stop-and-collect mode. In this mode the mutator is stopped whenever it runs out of allocatable memory, and control is given to the collector for reclamation of disused (i.e. inaccessible) cells. In some application areas (e.g. real-time or interactive systems) these possibly lengthy interruptions are not tolerable. This has lead to the development of non-stop versions of these algorithms originally suggested by Minsky [Knu68 (p. 417)] and Bobrow [Bob 66]. They may be classified in three families addressing different computational settings:

Parallel collectors: They are designed to allow the mutator and the collector to run as independent processes on possibly distinct processors sharing the same memory. Variations include running several cooperating mutators and/or processors. Here the main issue is to reduce or eliminate explicit synchronization and critical paths, to avoid slowing down the processes. Examples include [Ste 75, KunS 77, DijLMSS 78].

Real-time collectors: These collectors are organized to guarantee an upperbound on the time taken by any elementary cell manipulation instruction. This is achieved by evaluating the total time cost of a collection and spreading it more or less evenly over some of the cell manipulation primitives (mainly allocation primitives). Examples include [Bak 78, Bro 84, Yua 86]. However, it is probably impractical to consider real-time mark-and-sweep non-compacting collectors for varisized cells, because of the unpredictability of the time cost of cell allocation from a varisized free-list, and because of the unpredictability of the space cost of memory fragmentation. This remark also apply to the mixed strategy algorithm presented in this paper.

Incremental collectors: In incremental collectors, the mutator and the collector are two processes running on the same processor. In this case the problem is only to ensure that the collector may be resumed whenever the mutator is interrupted (for lack of memory or for any other reason), and that the collector may be quickly interrupted when the mutator is ready to resume computation. An important issue may be to control the collector so as to prevent useless collections that would waste the computing resources.

For all three types of non-stop collectors, data organization and modification must be carefully set-up so that the mutator and the collector do not confuse each other when modifying the same data structures.

In practice, the same techniques (with small variations) seem applicable to all three cases. For example, Wadler [Wad 76] shows how a parallel collector may be simplified into an incremental one. Similarly in [RepG 86], Brooks' variation of Baker's real-time algorithm is implemented as an incremental collector. Such a simplification into an incremental collector is rather straightforward for all parallel or real-time collectors.

Real-time algorithms have been proposed for both mark-and-sweep [Yua 86] and copy [Bak 78, Daw 82, Bro 84] algorithms, with essentially the same abstract organization. For parallel versions, most of the synchronization issues have been solved often elegantly in a succession of papers [Ste 75, Wad 76, KunS 77, DijLMSS 78] leading to versions of mark-and-sweep without explicit synchronization [KunS 77, DijLMSS 78]. However a parallel collector still requires some synchronization when logical cells have to be moved in storage, either for compaction [Ste 75, NewSW 83] or because it is a copy collector [BekCRU 86] (see also the multiprocessor real-time collector of Multilisp [Hal 84]). In [NewSW 83] a solution is offered that puts the burden of synchronization almost exclusively on the mutator.

Finally, it should be noted that a realistic (i.e. efficient) implementation of non-stop collection algorithms usually implies the use of specialized or microcoded hardware [Ste 75, Bak 78, Moo 84], though some proposals have been made for time-space trade-offs on vanilla hardware [BalS 83, Bro 84, NorR 87]. An incremental copy collection algorithm on stock hardware is proposed in [EllLA 88]. It makes use of the virtual memory hardware to detect mutator references to unvisited cells, but this technique is applicable only to copy collectors.

	Mark and Sweep	Copy
Stop-and-collect	McCarthy 1960	Minsky 1963 Fenichel & Yochelson 1969
Incremental	Wadler 1976	Reppy & Gansner 1986 Ellis, Li & Appel 1988
Real-time	Yuasa 1986	Baker 1978 Dawson 1982 Brooks 1984
Parallel	Steele 1975 Wadler 1976 Kung & Song 1977 [DijLMSS 1978]	Bekkers, Canet, Ridoux & Ungaro 1986

Figure 3: Examples of the various types of tracing garbage collectors.

2.5 Drawbacks of existing algorithms.

In either stop-and-collect or non-stop version, existing algorithms have some efficiency drawbacks. For copy collectors, the main problem is that half the memory is reserved for the copying process, and thus is lost for the mutator computation.

Mark-and-sweep collectors have a time cost proportional to the size of the available memory, rather than to the accessible memory as copy collectors. The collection overhead may thus be excessive when the mutator uses only a small part of this memory. Mark-and-sweep also requires some extra memory for marking bits and trace stack, though much less than copy collectors.

Another problem of mark-and-sweep collectors is that memory is fragmented in small chunks, harder to allocate, and sometimes too small to be useful. Compaction is possible in stop-and-collect mode, but it is usually complex and costly [Coh 81, CohN 83], and the necessary pointer relocation is considered so time expensive in [GotTHSI 79] as to justify the use of specialized hardware. In non-stop mode, one may use a *two-pointer* (also called *two-finger* or *folding)* compaction algorithm for cells of fixed size [Har 64, Ste 75, NewSW 83].

For varisized cells, other existing compaction algorithms for mark-and-sweep collection [HadW 67, Weg 72a, LanW 72, FitN 78] are harder to adapt to non-stop mode because they are all based on a *sliding* technique that moves used cells to one end of the available memory [Coh 81]. Cell sliding is usually performed after computing a *break-table* that gives the correspondence between old and new locations of cells. The difficulty is that some cells are moved on the former location of other accessible cells. This prevents storing forwarding pointers in the former location of moved cells, and thus the mutator executing in parallel cannot know where they have been moved. A rather costly solution might be to have the mutator access cells through the break-tables that give their new positions. A better solution is to access all cells through an indirection table similar to the object table used in many object-oriented systems [BalS 83, NorR 87]. Then one only has to update this unique reference when the object is moved for compaction. Though apparently workable, this approach does not seem to have been tried by anyone.

Non-stop adaptation seems still harder for the sliding *backchaining* (or *threading)* algorithms [Tho

76, DewM 77, Han 77, Mor 78, Jon 79], since they substantially modifies the logical organization of the accessible cells during the compaction process. This is equally true for algorithms based on some variation of the Deutsch-Schorr-Waite tracing algorithm [SchW 67, Weg 72b].

3 A mixed strategy

The aim of the mixed strategy collection algorithm we are about to describe is to have the two implementation strategies (i.e. mark-and-sweep and copy) compensate each other's drawbacks.

We shall first present a stop-and-collect version, and then indicate how to produce non-stop version of the algorithm.

The stop-and-collect version is interesting as a fairly cheap strategy to fight memory fragmentation, without the cost of extra passes on the memory for compaction.

The non-stop version is to our knowledge the only existing incremental compaction strategy that does not require limitation of mutator memory to half the available memory (i.e. use of copy collection) when varisized cells are allocated.

3.1 Stop-and-collect with incremental compaction

The fundamental idea of our algorithm is to divide the memory into two areas that are simultaneously collected, but with different collection strategies: copy and mark-and-sweep. By moving the copy-collected area with every collection cycle, we successively compact different parts of the memory. The free memory necessary to perform copy collection must be only as large as the copy-collected area, rather than half the available memory in classical copy collection. The trade-off between the size of the copy area and the compaction speed will be made clear below.

We first describe a simple static version of this approach. We assume that the available memory is divided into n+1 segments of equal size indexed from 0 to n. For each collection cycle two contiguous segments indexed i and i+1 are distinguished. The first one is supposed to be free storage at the beginning of the cycle and will be called *to-space* for this cycle. The next segment indexed i+1 will be called *from-space*, and the rest of the memory is called *ms-space* (for mark-and-sweep space). We also assume that we have all the extras required for some implementation of a mark-and-sweep collector over the whole memory, such as mark bits or extra space for storing the untraced set. For expository simplicity, we shall assume that mark-and-sweep is implemented by a mark bit and a tracing stack, though the actual details are not relevant.

Garbage collection proceeds as described in our abstract presentation, but with a mixed representation for the visited set and the untraced set. Visited cells are recognized by a mark bit when they are located in ms-space, and by being copied in to-space when they are located in from-space. Similarly untraced cells in ms-space are known by keeping them in a tracing stack (or any other mark-and-sweep technique), while untraced cells in to-space (i.e. originally in from-space at cycle beginning) are located in to-space according to some variation on the techniques due to Cheney [Che 70] for copy collection. When processing cells in the untraced set, it is advisable to give priority to the "mark-and-sweep" ones so as to limit the growth of the tracing stack.

As usual, we store a forwarding pointer in the old location of any cell that is moved from from-space to to-space. Any subsequently encountered pointer to the old location of that cell is updated by means of this forwarding pointer. Pointers to be updated may occur in ms-space and in to-space.

At the end of the cycle, all accessible cells are either marked or collected in a contiguous area of to-space. Then free memory is collected as follows:

- from-space is completely free and is kept to serve as to-space in the next collection cycle;
- unmarked cells from ms-space are collected into a free-list of available memory fragments;

- there usually is a remaining fragment of to-space that has not been used; it corresponds to inaccessible cells in from-space that have not been copied; this fragment is added to the free-list.

At the end of the collection cycle, the computation of the mutator is resumed with allocation from the free-list, until it runs out of memory. Then a new collection cycle is initiated, with segment i+1 (formerly from-space) as to-space, segment i+2 as from-space, and the rest of memory as ms-space. Of course the memory must be seen as cyclic, and segment i+2 must be replaced by segment 0 when i=n (i.e. segment indices are taken modulo n+1). The idea of a copy area moving through a cyclic memory has been first proposed (to our knowledge) by [BekCRU 86], but for full copy collection.

The adjacency of from-space and to-space has been assumed only to simplify the presentation, but it plays no role in this non-adaptive version of the algorithm. It is actually possible to choose from-space according to a more sophisticated strategy that depends on cells behavior. For example, from-space could be chosen as the most fragmented memory segment according to blocks found in the free-list. If collection is caused by a request for an unusually large block, from-space could be chosen as the segment containing the largest amount of free storage, even when it is not fragmented.

3.2 Specific Aspects

3.2.1 Memory compaction

The proposed algorithm does not usually compact all unused storage into a single piece, and thus it still requires a free-list of available fragments and an allocation strategy. However, it effectively fights memory fragmentation by periodically recompacting into a single larger piece all the small free fragments that may have been created in any given memory segment. The compaction process requires no extra passes on the list structures, unlike all compaction schemes for mark-and-sweep collection [Coh 81]. There is a memory cost which is the space required for copying, i.e. the free segment used as to-space.

There are n used memory segments, and one is compacted during each collection cycle. Thus the time required to recover the memory lost by fragmentation into very small chunks is on the order of n/2 (at most equal to n) collection cycles. The choice of n must be a trade-off between space losses due to the free segment for copying and space and time losses due to fragmentation (time losses may come from longer searches through the free-list). In a later section, we show how n may be dynamically adapted to the available resources and to the mutator requirements.

3.2.2 Free-list management

Our algorithm requires a structure (free-list) that keeps track of available memory fragments, and a strategy to allocate from these fragments when a new cell is requested by the mutator. To the authors' knowledge, the available literature on analysis, simulation or actual measurements of allocation strategies does not apply to our case for the following reasons:

- it is usually assumed that block freeing is a regular (stochastic) process [Sho 75, KorK 85] (caused by explicit freeing request of the mutator), rather than a periodical and more massive event as is the case with a non-compacting garbage collector. As a consequence the free-list behaves differently than it would with a collector: there is no coalescing to be expected as in [Sho 75]. There is also an increase of the complexity of the data structure used to support both the allocation strategy and the coalescing of adjacent free blocks [Ste 83, KorK 85].
- memory fragmentation is often emphasized over allocation time, while we are mainly interested in the latter for our algorithm.
- when time costs are analyzed, they include both allocation and coalescing, and the latter is not relevant in our case.
- published results are often inconclusive, when not contradictory, and seem to depend highly on block size distribution.

We believe that a best-fit strategy, similar to that proposed in [KorK 85], should be the best solution. Even in less favorable context than ours, its seems to perform generally well [Sho 75, KorK 85] with respect to fragmentation. The increased fragmentation (e.g. compared to first-fit) for which it is sometimes faulted is not an important issue in our case since all memory areas are ultimately compacted. With a proper structure of the free-list (as in [KorK 85]), allocation should be very fast, since blocks are sorted by size. The major cost of coalescing adjacent blocks is absent, since this work is performed by the collector. Furthermore, the construction of the structure serving as free-list (including the sorting of blocks) may be performed easily in parallel with the mutator in the case of non-stop collectors.

3.2.3 Data linearization and compression

The data *linearization* properties [Che 70, Cla 76, Coh 81] of our algorithm, though not as good as that of pure copy collection, still improve code locality and the effectiveness of list compression techniques (cdr-coding). Cdr-coding may be implemented in non-linearized structures as in Interlisp-D and in the FLATS machine [GotTHSI 79], but linearization has been shown to improve its performances considerably since the number of cdr pointers that may be eliminated may increase from 69.5[BobC 79]. Following the coding scheme and calculations described in [BobC 79] with 16 bit addresses and 2 bit cdr-coding, the average cost of a cell is 32 bits without cdr-coding, 29 bits with cdr-coding but no linearization (thus about 10for a given memory size), and 18.9 bits with cdr-coding and linearization (corresponding to a 69of cells). Our linearization is probably not as good as the above 97collection cycles between two passages of the copy area over a given memory location, but it should still perform much better than no linearization at all. Since linearization is not paid with half the memory as in pure copy collectors, our algorithm should permit the most effective use of cdr-coding to improve memory productivity.

Linearization and list compression may even be further improved by having to-space larger than from-space, so as to be able to copy complete lists in to-space even when they have some part originally belonging to ms-space. This is possible because the invisible forwarding pointers needed when copying cells are permitted anywhere as a standard feature in compressed lists [LiH 86], where there are necessary for performing *rplacd* operations on compressed cells. Thus invisible forwarding pointers are also usable in ms-space when lists are cdr-coded.

3.3 Non-stop implementations of the algorithm

Non-stop implementations of the mixed-strategy collector may be derived from existing non-stop collectors by respecifying them abstractedly, as we did above, and then by implementing the visited and untraced sets with a mixed representation as in the stop-and-collect case. Care must be taken not to put free fragments located in from-space in the free-list (unless allocation of unvisited cells is permitted as in [Daw 82] to reduce floating garbage and possibly improve linearization).

As noted in [DijLMSS 78 (p. 969)], the mutator must participate to some extent in the collection activity. The choice of the mutator primitive operations (for example some subset of *car, cdr, cons, eq, rplaca* and *rplacd* in the case of a Lisp mutator) on which the overhead of this participation is imposed varies according to authors. The nature of the participation varies as well. These variations are due to the complexity of the correctness proof [KunS 77, DijLMSS 78], the constraints of actual implementation choices, and efficiency considerations such as relative frequency of primitives, linearization of data [Che 70, Daw 82], and limitation of floating garbage [Wad 76, Daw 82, HicC 83] (i.e. limitation of the number of garbage cells disused during a collection cycle, that have to wait till the end of the next collection cycle before being collected).

On stock hardware, it is possible to combine Brooks' systematic forwarding pointers [Bro 84] with our mixed strategy. This may be too costly (compared to pure mark-and-sweep) for small cells. It may however be cost effective for compacting areas containing large cells (arrays or code) where the space overhead is small and the cost of memory fragmentation is high.

4 An adaptive version of the algorithm

4.1 The basic adaptive algorithm

When the copy segment (i.e. from-space or to-space) has a fixed size as in the algorithms described above, it is necessary to ensure that no cell is located across a segment border. Otherwise copying such a cell into to-space might overflow it if all other cells in from-space are accessible and thus copied; alternatively, not copying the cell would prevent complete freeing of from-space. When very large cells are allocated, this may not be an acceptable limitation, since it increases the memory fragmentation and the complexity of free-list management. Very large cells are quite common for data arrays, and even more for storing the code of compiled functions (which may sometimes be discarded after being called, or may be modified and recompiled)

This problem is solved by an adaptive version of our algorithm, that can change dynamically the size of the copy segment to adapt to varying requirements of the mutator and varying constraints of the running environment.

In the adaptive version, the memory is still viewed as circular as in [BekCRU 86], but it is divided in only three segments at the beginning of the collection cycle: from-space, to-space, and ms-space (one segment is usually physically cut in two parts because it extends across the top of the memory to its bottom). The only constraint now is that to-space and from-space have the same size (to-space may actually be larger), and that to-space is free at the beginning of the cycle, as described in figure 4a. The collection proceeds as in the previous versions of the algorithm, resulting in a memory configuration (figure 4b) where from-space is free, to-space is partially used by compacted data but (usually) still has a free sub-area contiguous to from-space, because inaccessible cells in from-space have not been copied.

Thus at the end of the cycle, we have a contiguous free area which is usually larger than the former size of from-space. We can use any leftmost part of it to carve the new to-space for the next cycle, and have a new from-space with the same size immediately to its right (left and right are meant to refer to figure 4). We can therefore decrease the size of the copied segment after each cycle, or increase it within the limits of the memory freed from the segment copied in the preceding cycle. The unused part of the free area is appended to the free-list for allocation to the mutator. In the case of the non-stop version, all free fragments located in the new from-space may have to be removed from the free-list before initiating a new cycle (figure 4c).

As in the non-adaptive version, there is no adjacency requirement on from-space and to-space. However, if they are not adjacent, there may be no extra free block next to a just collected from-space to permit its growth into a larger to-space for the next cycle. This aspect is to be considered too when choosing from-space at the beginning of a cycle.

4.2 Adaptive stop-and-collect version

In the stop-and-collect implementation of the algorithm, the size of the copy segment should be just big enough so that the cost of to-space does not exceed the gains from the reduced fragmentation, since any reduction of the computationally usable memory results in more frequent collection. It is even possible to run pure mark-and-sweep collection for some time, and only restart partial copy collection from some large free segment in the free-list when memory fragmentation is judged too important, and only until it is reduced.

However, when the size of the accessible memory becomes much smaller than the available memory, it may become more efficient to increase the role of copy collection over that of mark-and-sweep. The reason is that the cost of copy collection is proportional only to the number of accessible (copied) cells, while the cost of mark-and-sweep collection is proportional to the size of the whole memory segment collected by mark-and-sweep (plus the cost of allocation from a free list in the case of varisized cells). Another factor is the relative per-cell costs of marking, sweeping, and copying.

The actual trade-off equilibrium point is dependent on the amount of memory accessible by the mutator, and thus varies with time. Dynamic computation of this trade-off point may be based on a theoretical

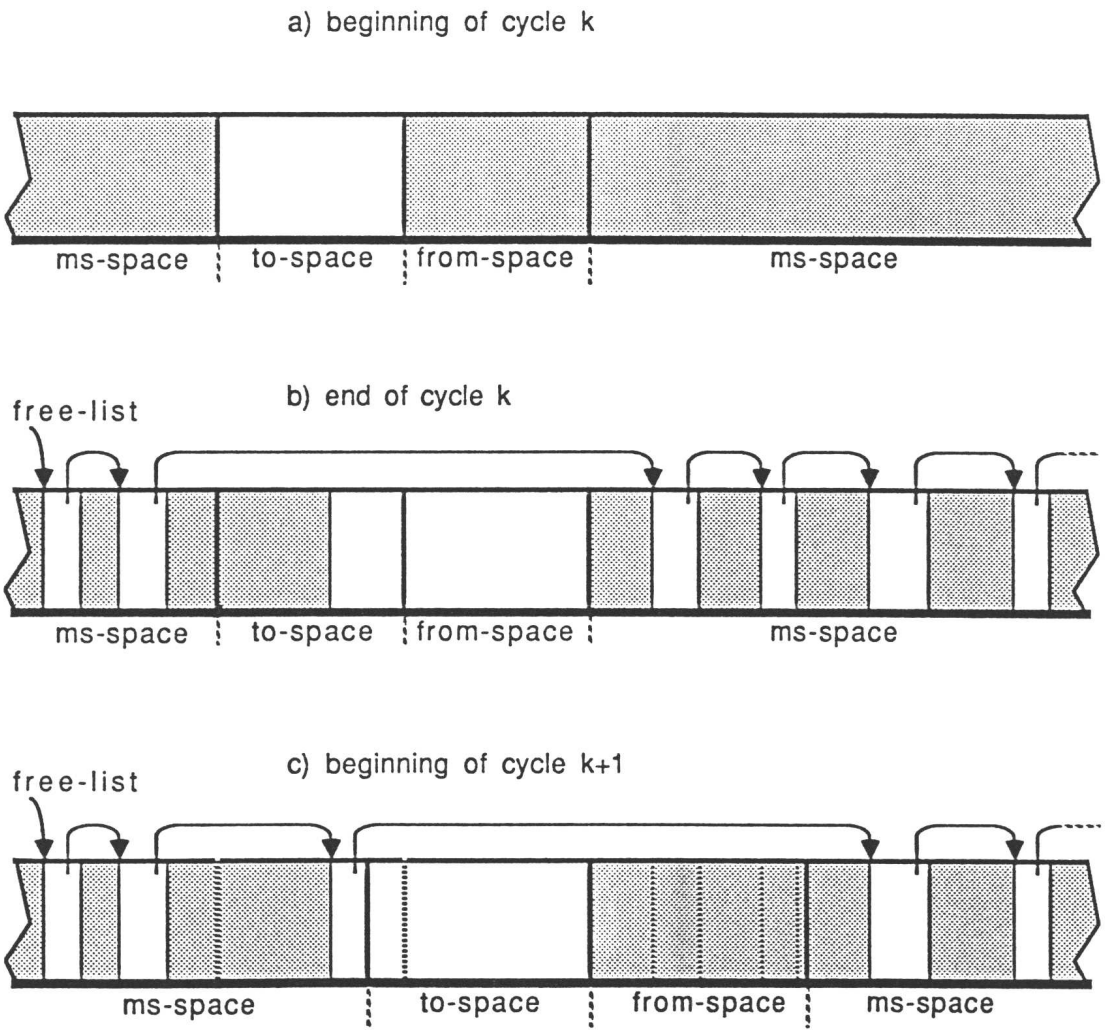

Figure 4: The adaptive non-stop collector

analysis of the algorithm, or on experimental measured data stored in a table.

4.3 Adaptive non-stop version

The analysis of trade-offs between copy and mark-and-sweep is somewhat different for non-stop implementations of the adaptive algorithm, though several of the above remarks are still relevant. The additional factors are the relative speed of the mutator and the collector, and the mutator overhead. Though we shall not give a detailed analysis of the behavior of the collection process (see for example [Wad 76, HicC 84]), we can make some qualitative remarks.

The main objective is to have the mutator run as fast as possible. Given the relative speeds of the mutator (for allocation) and of the collector (to terminate a cycle), we must reduce as much as possible (or eliminate) the time the mutator spends waiting for the end of a collection cycle because it has run out of memory. Thus the free-list must be made big enough at the beginning of each collection cycle to avoid this situation, which in turn may limit the amount of memory that may be used for copying, depending on the available and the effectively accessed memory.

Another factor that may justify reducing the role of copying *for parallel implementations* is the synchronization overhead that copying may impose on the mutator and the collector for protecting access to cells while they are being moved or updated [Ste 75, NewSW 83, Hal 84].

For incremental or real-time implementations, as for the stop-and-collect case, copy collection is cheaper than mark-and-sweep when only a small fraction of memory is actually used.

To maintain contiguity of from-space and the part of to-space left free, and to also improve linearization as suggested in [Daw 82], it is necessary to take some care in the way to-space is used for mutator induced copies. One possible solution is to perform copies due to the mutator primitives in the free-list rather than in to-space. Then marking bits have to be used in from-space to identify forwarding pointers.

5 Relation to multiple-area collectors

Division of the heap into several areas has been used in several systems. It is often based on the type of objects allocated in each area, for such purposes as easy determination of the type of objects [Con 74], improvement of data locality [NiwYHH 86], or different data management policies for different types [ChaDH 84]. Distinction between areas may also be based on variation in the collection strategy and/or periodicity for each area, particularly in the case of generation collectors [BroGS 82, LieH 83, Moo 84, Ung 84].

Our algorithm was partly inspired by a system of the former kind [ChaDH 84] in which distinct areas are collected simultaneously, with distinct though cooperating collection strategies. Owing to the unconstrained choice of the location of from-space, our algorithm may to some extent be used as a collector of the latter kind by focusing copy collection on areas where it is most useful.

To our knowledge, generation collectors are all based on copy collection which both compacts memory and naturally includes means to move cells between areas. For the same reasons, our algorithm is usable in that context, with the constraint that *graduation* from ephemeral to more stable storage [Moo 84, Ung 84] is done incrementally from the copy collected segment. Thus a cell that has survived enough collection to be eligible for graduation must still wait until it is in the copy collected area of a collection cycle.

6 A practical limitation

In an actual system, it is sometimes necessary to perform full memory compaction, without regard for cost or incrementality. This is for example the case in the following circumstances:

- despite a just completed collection cycle, the mutator needs a cell that exceeds the size of each available memory chunks, though not their total size;
- a core image of the running system is to be created on secondary storage: it is desirable to easily identify in large fragments the unused parts of memory so as to remove them from the core image and thus reduce the space requirements of that image.

If memory is limited (thus preventing the use of pure copy), and if varisized cells are being allocated (thus preventing the use of the two-pointer compaction algorithm [Har 64, Ste 75, NewSW 83]), it is then necessary to complement our algorithm with a compacting stop-and-collect algorithm.

The marking phase may be performed by our algorithm (without its sweep phase) without a copy area, or with the additional marking of the used part of to-space once the tracing phase is finished. Then only a *break-table* compacter [HadW 67, Weg 72a, LanW 72, FitN 78] has to be added to the system for full compaction purposes. It is also possible to use a complete threading collector [Tho 76, DewM 77, Han 77, Mor 78, Jon 79] that integrates both marking and compaction.

7 Conclusion

In stop-and-collect mode our algorithm is mostly useful because it is simpler and/or cheaper than other compacting algorithms, using less passes and/or memory. Its adaptiveness in a continuum from pure mark-and-sweep to pure copy collection minimizes collection costs according to the amount of memory actually used. It can also adapt the choice of the copy-collected area according to the fragmentation profile of the memory.

As a non-stop collector, this algorithm can compact varisized cells without restricting effective mutator use of the memory to half its actual size. This property is particularly important for real-time collectors since, because of the time cost and unpredictability of paging, large virtual memory may not always be acceptable for real-time applications [Bak 78], and real memory is usually relatively small.

We believe that our mixed strategy algorithm would be especially adapted for collection in the old generation of a generation collector for the following reasons:

- varisized free-list allocation cost is amortized over longer lived cells,
- copy collection is less efficient when the proportion of garbage is lower,
- overhead due to interaction with the mutator (i.e. contention for cell access in the copy part of a parallel mixed collector) is reduced due to the lesser average activity of old cells.

The same mixed collector may also be used, in pure copy mode, for the young generations.

E. Spir has implemented the adaptive algorithm in stop-and-collect mode for the Le-Lisp system [ChaDH 84].

Acknowledgements

The work presented here was initially motivated by a private explanation of his algorithm [BekCRU 86] given by Yves Jean Bekkers to the first author. The idea of the circulating partial copy collection grew out of the analysis of this algorithm. The second author suggested its combination with a mixed strategy collection on static areas that he had implemented for the garbage collector of Le-Lisp [ChaDH 84].

We benefited from discussions with Eric Spir and the members of the Le-Lisp group: Jérôme Chailloux, Matthieu Devin, Greg Nuyens and Bernard Serpette, and also with Mats Bengtsson.

This work has been partly supported by Esprit Project 348: *Generation of Interactive Programming Environments (GIPE).*

This paper is a slightly revised version of a paper with the same title that appeared in the proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques (St. Paul, Minnesota, June 1987), published as: SIGPLAN Notices, Vol. 22, No. 7, July 1987, pp. 253-263.

References

[Arn 72]
Arnborg, S., Storage Administration in a Virtual Memory Simula System, *BIT,* Vol. 12, pp. 125-141, 1972

[Bak 78]
Baker, H.G., List Processing in Real Time on a Serial Computer, *Comm. ACM,* Vol. 21, No. 4, pp. 280-294, April 1978.

[BalS 83]
Ballard, S., and Shirron, S., The Design and Implementation of VAX/Smalltalk-80, in *Smalltalk-80: Bits of History, Words of Advice,* G. Krasner (ed.), Addison Wesley, pp 127-150, September 1983.

[BekCRU 86]
Bekkers, Y., Canet, B., Ridoux, O., and Ungaro, L., MALI: A Memory with a Real-time Garbage Collector for Implementing Logic Programming Languages, *Proc. of the 3rd IEEE Symp. on Logic Programming,* Salt-Lake City (Utah), September 1986.

[Ben 84]
Ben-Ari, M., Algorithms for On-the-fly Garbage Collection, *ACM Trans. on Programming Languages and Systems,* Vol. 6, No. 3, pp. 333-344, July 1984.

[Bob 66]
Bobrow, D.G., Storage Management in LISP, *Proc. of the IFIP working Conf. on Symbol Manipulation Languages and Techniques,* North-Holland (1968), D.G. Bobrow (ed.), pp. 291-301, Pisa (Italy), September 1966.

[BobC 79]
Bobrow, D.G., and Clark, D.W., Compact Encodings of List Structure, *ACM Trans. on Programming Languages and Systems,* Vol. 1, No. 2, pp. 266-286, October 1979.

[BobM 67]
Bobrow, D.G., and Murphy, D.L., Structure of a LISP System Using Two-Level Storage, *Comm. ACM,* Vol. 10, No. 3, pp. 155-159, March 1967.

[BroGS 82]
Brooks, R.A., Gabriel, R.P., and Steele, G.L. Jr., S-1 Common Lisp Implementation, *Conf. Record of the 1982 ACM Symp. on Lisp and Functional Programming,* Pittsburgh (Pennsylvania), pp. 108-113, August 1982.

[Bro 84]
Brooks, R.A., Trading Data Space to Reduce Time and Code Space in Real-time Garbage Collection on Stock Hardware, *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming,* Austin (Texas), pp. 256-262, August 1984.

[ChaDH 84]
Chailloux, J., Devin, M., and Hullot, J.M., Le-Lisp: A Portable and Efficient Lisp System, *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming,* Austin (Texas), pp. 113-122, August 1984.

[Che 70]
Cheney, C.J., A Nonrecursive List Compacting Algorithm, *Comm. ACM,* Vol. 13, No. 11, pp. 677-678, November 1970.

[Cla 76]
Clark, D.W., An Efficient List Moving Algorithm using Constant Workspace, *Comm. ACM,* Vol. 19, No.6, pp. 352-354, June 1976.

[Coh 81]
Cohen, J., Garbage Collection of Linked Data Structures, *ACM Computing Surveys,* Vol. 13, No. 3, pp. 341-367, September 1981.

[CohN 83]
Cohen, J., and Nicolau, A., Comparison of Compaction Algorithms for Garbage Collection, *ACM Toplas,* Vol. 5, No. 4, pp. 532-553, October 1983.

[Col 60]
Collins, G.E., A Method for Overlapping and Erasure of Lists, *Comm. ACM,* Vol. 3, No. 12, pp. 655-657, December 1960.

[Con 74]
Conrad, W.R., *A Compactifying Garbage Collector for ECL's Non-Homogeneous Heap,* Tech. Report 2-74, Harvard Univ., Cambridge (Mass.), February 1974.

[Daw 82]
Dawson, J.L., Improved Effectiveness from a real time Lisp Garbage Collector, *Conf. Record of the 1982 ACM Symp. on Lisp and Functional Programming,* Pittsburgh (Pennsylvania), pp. 159-167, August 1982.

[DeuB 76]
Deutsch, L.P., and Bobrow, D.G., An Efficient, Incremental, Automatic Garbage Collector, *Comm. ACM,* Vol. 19, No. 9, pp. 522-526, September 1976.

[DewM 77]
Dewar R.B.K., and McCann, A.P., MACRO SPITBOL - a SNOBOL4 Compiler, *Software - Practice and Experience,* Vol. 7, No. 1, pp. 95-113, January 1977.

[DijLMSS 78]
Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., and Steffens, E.F.M., On-the-fly Garbage Collection: An Exercise in Cooperation, *Comm. ACM,* Vol. 21, No. 11, pp. 966-975, November 1978.

[EllLA 88]
Ellis, J.R., Li, K., and Appel, A.W., *Real-time Concurrent Collection on Stock Multiprocessors,* Systems Research Center Report 25, DEC, Palo-Alto (California), February 1988.

[FenY 69]
Fenichel, R.R., and Yochelson, J.C., A LISP Garbage Collector for Virtual-memory Computer Systems, *Comm. ACM,* Vol. 12, No. 11, pp. 611-612, November 1969.

[FitN 78]
Fitch, J.P., and Norman, A.C., A note on Compacting Garbage Collection, *The Computer Journal,* Vol. 21, No. 1, pp. 31-34, February 1978.

[GotTHSI 79]
Goto, E., Tetsuo, I., Hiraki, K., Susuki, M., and Inada, N., FLATS, A machine for Numerical, Symbolic and Associative Computing, *Conf. Proc. of The 6th Ann. Symp. on Computer Architecture,* Philadelphia, pp. 102-110, April 1979.

[HadW 67]
Haddon, B.K., and Waite, W.M., A Compaction Procedure for Variable-length Storage Elements, *The Computer Journal,* Vol 10, pp. 162-165, August 1967.

[Hal 84]
Halstead, R.H. Jr., Implementation of Multilisp: Lisp on a Multiprocessor, *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming,* Austin (Texas), pp. 9-17, August 1984.

[Han 69]
Hansen, W.J., Compact List Representation: Definition, Garbage Collection, and System Implementation, *Comm. ACM,* Vol. 12, No. 9, pp. 499-507, September 1969.

[Han 77]
Hanson, D.R., Storage Management for an Implementation of SNOBOL4, *Software - Practice and Experience,* Vol. 7, No. 2, pp. 179-192, March 1977.

[Har 64]
Hart, T.P., and Evans, T.G., Notes on Implementing Lisp for the M 460 computer, in *The Programming Language LISP,* E.C. Berkeley and D.G. Bobrow (eds.), M.I.T., Cambridge (Mass.), 4th printing, 1974.

[Hib 80]
Hibino, Y., A Practical Parallel Garbage Collection Algorithm and its Implementation, *Conf. Proc. of The 7th Ann. Symp. on Computer Architecture,* Rennes (France), May 1980, in *SIGARCH Newsletter,* Vol. 8, No. 3, pp. 113-120.

[HicC 64]
Hickey, T., and Cohen, J., Performance Analysis of On-the-Fly Garbage Collection, *Comm. ACM,* Vol. 27, No. 11, pp. 1143-1154, November 1984.

[Jon 79]
Jonkers, H.B.M., A Fast Garbage Compaction Algorithm, *Inform. Processing Letters,* Vol. 9, No. 1, pp. 26-30, July 1979.

[Knu 68]
Knuth, D.E., *The Art of Computer Programming, Vol. 1, Fundamental Algorithms,* Addison-Wesley, Reading, Mass., 1968.

[KorK 85]
Korn, D.G., and Kiem-Phong, V., In Search of a Better Malloc, *Summer Conf. Proc. of the Usenix Association,* Portland (Oregon), pp. 489-506, June 1985.

[KunS 77]
Kung, H.T., and Song, S.W., An Efficient Garbage Collection System and its Correctness Proof, *Proc. of IEEE 18th Symposium on Foundations of Computer Science,* Providence (R.I.), pp. 120-131, October-November 1977.

[LanW 72]
Lang, B., and Wegbreit, B., *Fast Compactification,* Rep. 25-72, Harvard Univ., Cambridge (Mass.), November 1972.

[LiH 86]

Li, K., and Hudak, P., A New List Compaction Method, *Software - Practice and Experience,* Vol. 16, No. 2, pp. 145-163, February 1986.

[LieH 83]
Liebermann, H., and Hewitt, C., A Real-time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM,* Vol. 26, No. 6, pp. 419-429, June 1983.

[McC 60]
McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, *Comm. ACM,* Vol. 3, No. 4, pp. 184-195, April 1960.

[Min 63]
Minsky, M.L., *A Lisp Garbage Collector Algorithm using Serial Secondary Storage,* Memo 58, M.I.T. A.I. Lab., M.I.T., Cambridge, Mass., December 1963.

[Moo 84]
Moon, D.A., Garbage Collection in a Large Lisp System, *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming,* Austin (Texas), pp. 235-246, August 1984.

[Mor 78]
Morris, F.L., A Time- and Space-Efficient garbage compaction algorithm, *Comm. ACM,* Vol. 21, No. 8, pp. 662-665, August 1978.

[NewSW 83]
Newman, I.A., Stallard, R.P., and Woodward M.C., A Parallel Compaction Algorithm for Multiprocessor Garbage Collection, *Proc. of Internat. Conf. on Parallel Computing 83,* Feilmeier M., Joubert J. and Schendel U. (eds.), Elsevier Science Publishers B.V. (North-Holland), 1983.

[NiwYHH 86]
Niwa M., Yuhara M., Hayashi K., and Hattori A., Garbage Collection with Area Optimization for Facom ALPHA, *Proc. of the 31st IEEE Comp. Soc. Int. Conf., Spring CompCon 86,* San Francisco (Cal.), A. G. Bell (ed.), pp. 235-240, March 1986.

[NorR 87]
North, S.C., and Reppy, J.H., Concurrent Garbage Collection on Stock Hardware, *Proc. of Conf. on Functional Languages and Computer Architecture,* G. Kahn (ed.), Portland (Oregon), pp. 113-133, September 1987.

[RepG 86]
Reppy, J.H., and Gansner, E.R., A Foundation for Programming Environments, *Proc. of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments,* P.B. Henderson (ed.), Palo-Alto (California), December 1986, published as *SIGPLAN Notices,* Vol. 22, No. 1, pp. 218-227, January 1987.

[Rud 86]
Rudalics, M., Distributed Copying Garbage Collection, *Proc. of the 1986 ACM Conf. on Lisp and Functional Programming,* Cambridge (Mass.), pp. 364-372, August 1986.

[SchW 67]
Schorr, H., and Waite, W.M., An Efficient Machine-independent Procedure for Garbage Collection in various List Structures, *Comm. ACM,* Vol. 10, No. 8, pp. 501-506, August 1967.

[Sho 75]
Shore, J.E., On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies, *Comm. ACM,* Vol. 18, No. 8, pp. 433-440, August 1975.

[Ste 75]
Steele, G.L. Jr., Multiprocessing Compactifying Garbage Collection, *Comm. ACM,* Vol. 18, No. 9, pp. 495-508, September 1975.

[Ste 83]
Stephenson, C.J., Fast Fits: New Methods for Dynamic Storage Allocation, *Ninth ACM Symp. on Operating Systems, SIGOPS Review,* Vol. 17, No. 5, pp. 30-32, 1983.

[Tho 76]
Thorelli, L.E., A Fast Compactifying Garbage Collector, *BIT,* Vol. 16, No. 4, pp. 426-441, 1976.

[Ung 84]
Ungar, D., Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *Proc. of ACM SIGSOFT/SIGPLAN Conference on Practical Programming Environments,* Henderson P.B. (ed.), pp. 157-167, April 1984.

[Wad 76]
Wadler, P.L., Analysis of an Algorithm for Real Time Garbage Collection, *Comm. ACM,* Vol.

19, No. 9, pp. 491-500, September 1976.

[Weg 72a]
Wegbreit, B., A Generalized Compactifying Garbage Collector, *The Computer Journal,* Vol. 15, No. 3, pp. 204-208, August 1972.

[Weg 72b]
Wegbreit, B., A Space-Efficient List Structure Tracing Algorithm, *IEEE Transactions on Computers,* Vol. C-21, No. 9, pp. 1009-1010, September 1972.

[Yua 86]
Yuasa, T., *Realtime Garbage Collection on General-purpose Machines,* Research Report, Research Institute for Mathematical Sciences, Kyoto University, Japan, February 1986.

History and copyright

A first version of this papers was prsented at the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, St. Paul, Minnesota, Juin 1987. It was later slightly expanded into this version, which was prepared for the Second Franco-Japanese Symposium on "Programming of Future Generation Computers", Cannes (France), November 9-11, 1987. It was published in the proceeding edited by K. Fuchi and L. Kott and published by North-Holland / Elsevier Science Publishers B.V. (1988), ISBN: 0 444 70526 0, pages 163-182.

These first two versions were originally written in Troff, with the figures hand pasted. In May 2003, the first author translated the Troff source code of the second expanded version into this LaTeX variant (with very minor presentation changes – which explains the unusual style of this LaTeX file – and the correction two typos), and included the figures. Unfortunately, the original figures, produced on a Macintosh, were no longer usable (lost resource files). They had to be scanned from a paper copy.

The sources contains numerous additionnal informal comments, originally intended for the personnal use of the authors, which may or may not be useful to readers. One version of the LaTeX source produces the papers with these comments.

Since the officially published versions have been unavailable for some time in any form (electronic or printed), you may (and are even encouraged to `:-)` use, reproduce and disseminate this one, according to the Free Document Dissemination Licence (FDDL v1) available at
`http://pauillac.inria.fr/~lang/licence/v1/fddl.html` .

This paper and its source are available at (or around):
`http://pauillac.inria.fr/~lang/papers`
`http://www.datcha.net/bernard/papers`
`http://atoll.inria.fr/biblio`

At the date this Latex version is produced (May 2003), the authors can be reached at `Bernard.Lang@inria.fr` and `Francis.Dupont@enst-bretagne.fr` .