# Chapter 1

# Introduction

Automatic memory management is a collection of memory management techniques which free programmers from the chore of keeping track of which areas of memory are in use and which are available for re-use. The emphasis in this paper is on the automatic aspects of memory management, namely identifying and reclaiming unused storage rather than on techniques of memory management such as quick fit [Wein88] and the buddy system [Knut73]. This paper also covers such techniques as *cdr-encoding* which are concerned with compact representations of common LISP data structures.

The objective of this paper is to identify techniques suitable for use on a small, single user, real-memory workstation using a stock CPU (one with no hardware support specifically tailored for LISP). The assumption is that such workstations have limited amounts of real memory and no virtual memory capability. These restrictions mean that making effective use of memory is very desirable. In addition, since such workstations are typically operated interactively by a single user, long pauses for *garbage collection* during which the memory manager identifies and reclaims all unused storage are unacceptable.

# Chapter 2

# Memory Organization

The basic unit of memory in a LISP system is the *node*. A node is a number of consecutive words of memory which must contiguous. The words in the node may contain control information describing the node itself, pointers to other nodes and they any other kind of information. A node may be moved relative to other nodes, but words in a node may not move relative to each other.

All nodes have an associated *type*. The type of a node determines its size and determines which words in the node are pointers to other nodes and which are not. Type information is associated with nodes in one of three ways:

- Type information may be encoded as a bit field associated with every node [Betz86] or every word of memory [Moon84]. Since the average size of LISP nodes is between 10 and 20 bytes [ClarG77] [ClarG78], using a one byte *node header* to encode type information results in a memory overhead of between 5% and 10%. This figure can be reduced when there is already sufficient waste space in a node to encode type information. In general, node headers are a fairly simple and portable way to associate type information with LISP nodes.

- Type information may be encoded as a bit field within every pointer, identifying the type of node the pointer points at [Hans69], [Broo84]. This is possible either when there are spare bits in pointers which are ignored by the addressing hardware, or when the software masks the type bits out of pointers before using them as addresses. Typed pointers require no additional storage since they make use of otherwise wasted storage inside of pointers.

  Sweeping storage from one end to the other, is impossible using typed pointers since size information is associated with the node type information. Type (and size) information is available only when chasing a pointer to a node and not when constructing a pointer by adding a size to an existing pointer pointing to a node.

- Type information may be associated with nodes by locating all nodes of a given type in the same page (Brooks' *big bag of pages (BIBOP)* [Stee77]). If a page is a region

of memory which is a power of two words in length, the beginning of the page can be found by masking out the appropriate bits in a pointer to a node in the page. Type information can be associated with all nodes in a page either through a table associating page starting addresses with types or by encoding the type information in the first word of the page.

Using two bytes of memory at the beginning of every 1024 byte page to encode type information adds insignificantly to the memory cost of the LISP implementation. The fact that memory must be allocated to types in fixed blocks means that, on average, there will be one half block of memory wasted per data type. In a half Megabyte system using 1024 byte pages and using 20 data types, this represents a 2% memory overhead. This overhead can increase significantly if some nodes are a significant fraction of a page long and a page does not contain an integral number of nodes.

While its memory and processing requirements are small, the implementation of memory allocation and collection portions of a BIBOP system are quite complex. Difficulties arising from allocation of nodes which are almost as large as or larger than a page and from a non-contiguous address space are all more difficult to deal with in BIBOP than in other kinds of memory organization strategies.

On the other hand, since most LISP data types are of a fixed length (the exceptions being strings, vectors and cdr-encoded lists), and a single memory page in BIBOP contains nodes of only one type, many pages will contain fixed size nodes. Compaction of these fixed size nodes is trivial [Knut73] and can be accomplished very efficiently.

When a LISP implementation deals with a large number of types, a hybrid method may be used where either the BIBOP method or typed pointers are used for the most common node types and where node headers are used for the less frequently used node types.

# Chapter 3

# Identifying Unused Memory

All algorithms for identifying areas of memory which are available for re-use fall into one of two families: reference count schemes and tracing schemes.

Tracing schemes identify those memory locations which are still in use by walking the tree of pointers to nodes rooted at a small number of fixed locations known to the LISP system. These fixed locations usually include CPU registers and the LISP stack, thus guaranteeing that even nodes which are temporarily in use will be traced. As each node is encountered, it is identified as being in use, usually by turning on a *mark bit* in the node. Any node which is not encountered and marked during the trace of all nodes in memory is not in use.

Tracing schemes can use external bit maps to identify nodes which are in use, but such maps are rarely used in modern LISP systems because external bit maps occupy valuable storage. Space can almost always be found within the node structure for a mark bit. Tracing schemes which move trees of nodes (*lists*) during tracing do not require mark bits to identify nodes which are in use (see section 2 below).

Reference count schemes keep track of the number of pointers to a given memory location. When this count reaches 0 for any node, there are no pointers anywhere in the LISP system which point to the node and the node can be immediately re-used. Reference count schemes suffer from an inability to reclaim circular chains of pointers. To reclaim such circular list structures, reference count schemes may be augmented by periodic use of a tracing scheme.

## 3.1 List Tracing by Pointer Reversal

Probably the oldest scheme for tracing an arbitrarily linked set of nodes [Knut73] involves a stack of pointers to nodes containing at least one pointer to another node (Appendix A).

The problem with this method is that it requires a stack of pointers to nodes and current locations within those nodes. The size of this stack, in the worst case, is proportional to the number of nodes still in use.

This memory cost can be substantially reduced by distributing information from the stack amongst visted nodes in a technique called *pointer reversal* [Knut73] [SchoW67]. This technique effectively reverses the list structure of the list being traversed, thus allowing the

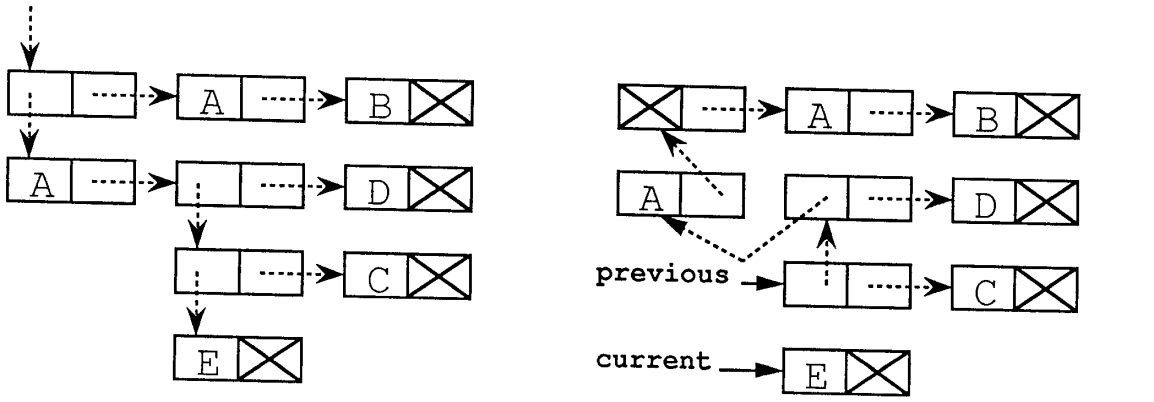



Figure 3.1: Tracing By Pointer Reversal

tracing routine to find the beginning of the list again after it has reached the end of the list. As the garbage collector works its way back to the beginning of the list, it reverses the list again, restoring the original list structure. This list reversal eliminates the need to store pointers to nodes on the garbage collector stack.

Even with pointer reversal however, there must be some mechanism to keep track of the current pointer location in a node which may contain many pointers. If most nodes contain at most two pointers as [SchoW67] assumes, a single bit can be reserved in these nodes to indicate which of the pointers have been traced. More generally, [Wegb72b] points out that position information is required only for those nodes containing many pointers. If a node contains only two pointers, this position information can be encoded in a single bit. If a node contains $N$ pointers, this information can be encoded in no more than $log_2 N$ bits. Room for these bits can either be reserved in each node with pointers, or these bits can be stored on a separate bit stack (Appendix B).

While these techniques may reduce the size of the trace stack by an order of magnitude, they do not eliminate the possibility of a stack overflow. The only way to cope with this possibility is to either:

- Encode the remaining stack information (the index of the current pointer in the node) within nodes. This requires $log_2(number\ of\ pointers\ in\ a\ node)$ bits per node.

- Use a technique like that of [Knut73] to recover from stack overflow. Knuth suggests that the oldest information in the stack simply be discarded when the stack overflows. This means that some some parts of the tree of nodes will remain untraced. To trace these nodes, the garbage collector must traverse all of memory and continue tracing all nodes which are marked as traced, but whose pointers point to untraced nodes.

If the latter method is used, pointer reversal cannot be used. Discarding stack information during a pointer reversal trace results in permanent damage to the structure of the node tree being traced.

A variant of the former method of encoding stack information in nodes, is to somehow identify the pointer in the node which has been reversed. This can be accomplished by reserving a bit in each pointer or by re-using a field in the pointer whose value can be re-created when the pointer is restored [Knut73]. The disadvantage of this mechanism is that it requires that each node be scanned for the reversed pointer every time the node is encountered. This increases the cost of marking a node from $O(N)$ to $O(N**2)$ where $N$ is the number of pointers in the node. This cost can be reduced by using a stack whenever possible and resorting to marking reversed pointers only when stack space is exhausted.

## 3.2 Tracing By Moving

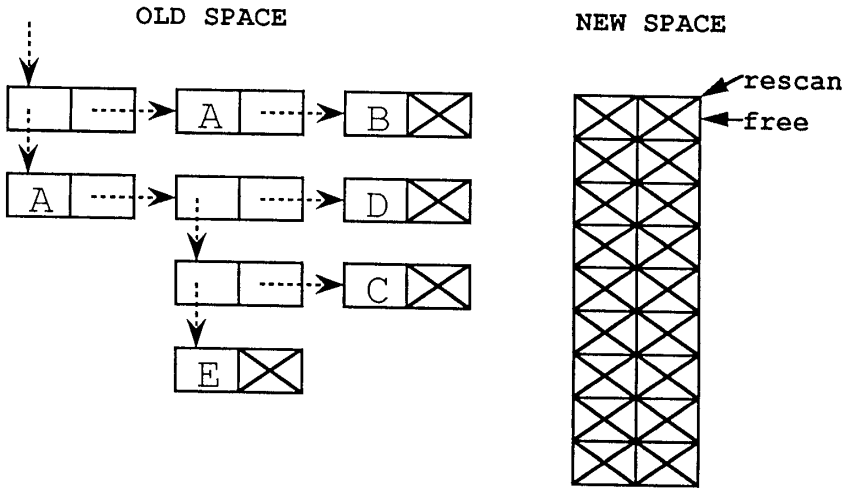
Some of the garbage collection techniques discussed in the next chapter require moving trees of nodes from one location to another in addition to tracing them. To guarantee that the moved set of nodes is connected in the same way as the original set, the move operation must carry out *pointer adjustment.* Pointer adjustment modifies every pointer in every moved node so that for all nodes $A$ and $B$ in the tree of nodes to be moved, and for all pointers $P$ in $A$, if $B'$ and $P'$ are the moved copies of $B$ and $P$ respectively, and $P->B$ before the move then $P'->B'$ after the move.

The next section provides many mechanisms for pointer adjustment. For now we are concerned only with the *forwarding pointer* mechanism used in an elegant algorithm invented by Cheney [Chen70]. A forwarding pointer is a pointer distinguishable from other pointers and which indicates the new location of a node. Forwarding pointers can be distinguished from other pointers either by some bit field in the pointer or by using the BIBOP method where pages containing moved pointers are marked as *new.* This latter method is used almost exclusively in tracing by moving algorithms since these algorithms are almost always used to move a tree of nodes into an entirely new set of memory pages. The use of forwarding pointers places a lower bound on the size of a node: every node must be large enough to contain a forwarding pointer.

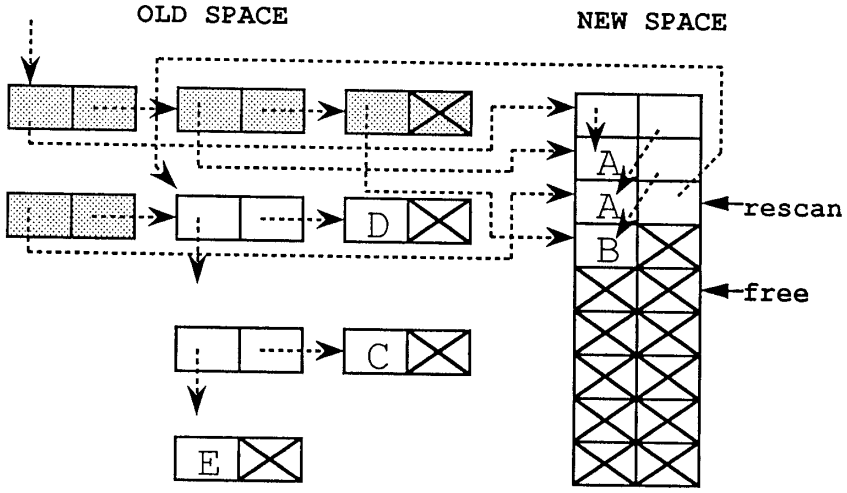
Cheney's algorithm allocates a number of new pages of memory called *new-space*, sufficiently large to hold all of the used nodes in the tree of nodes to trace. The set of pages containing this tree is called *old-space.* The algorithm maintains two pointers into new-space:

- *next-free* points to the beginning of a contiguous region of unused memory in new-space. This pointer initially points to the beginning of new-space and is advanced sequentially through new-space as nodes are moved to new-space.

- *rescan* points to the beginning of the contiguous region of nodes in new-space which have been moved, but whose pointers have not all been adjusted to point into new-space. This pointer initially points to the beginning of new-space, is advanced sequentially through new-space as pointers in moved nodes are adjusted, and never moved




a) original list structure




b) part way through
tracing by moving

Figure 3.2: Tracing by Moving

beyond the next-free pointer. When the rescan pointer meets the next-free pointer, the trace of the tree of nodes in old-space is complete.

The tracing algorithm begins by copying the root node from old-space to the beginning of new-space, replacing the root node in old-space with a forwarding pointer to the beginning of new-space and advancing the next-free pointer in new-space by the size of the node moved. The tracing algorithm then advances the rescan pointer until it meets the next-free pointer. For every node the rescan pointer passes in new-space, every pointer to old-space is examined. If the pointer in new-space points to a forwarding pointer, it is replaced with the forwarding pointer. If the pointer in new-space points to a node in old-space, that node is copied to the next-free location in new-space and next-free is advanced. The list is completely copied when the rescan and next-free pointers are identical. This algorithm is illustrated in Appendix C.

This algorithm visits every node to be traced exactly twice: once to copy it to new-space and once to adjust pointers in the node. A small optimization of Cheney's algorithm is presented in [Clar76] which exploits the fact that only those nodes containing more than one pointer need to be visited twice. Clark actually applies the algorithm only to nodes containing two pointers, but the idea is easily generalized to nodes containing many pointers.

Clark suggests maintaining list of nodes which must be visited twice rather than sequentially searching new-space for these nodes. The list is linked using the memory in the copy of the node in old-space. The copy of the node in old space is guaranteed to be big enough to hold both the forwarding pointer and the link, because only those nodes containing two or more pointers are added to the list for pointer adjustment. Nodes containing no pointers do not require pointer adjustment and nodes containing one pointer have their pointer adjusted as they are copied to new-space. The optimized algorithm can be seen in Appendix D.

Clark's modification performs better than the original since it visits twice only those nodes which need to be visited twice. In addition, since Clark's modification involves no sequential sweep of nodes in either old-space or new-space, it can be used with typed pointers. Cheney's original algorithm accesses nodes sequentially in new-space and so will not work with typed pointers.

## 3.3 Reference Counting

Reference counting schemes keep track of how many pointers there are to a given node by associating an integer, called the *reference count*, with each node. Every time a pointer is made to point to a node, the node's reference count is incremented. Every time a pointer is changed from pointing at one node to pointing at something else, the node's reference count is decremented. When a node's reference count reaches 0, the node can be immediately re-used.

The advantage to reference count schemes is that they update trace information continuously, rather than suspending the LISP system periodically while identifying and collecting unused memory. The two biggest disadvantages are:

- At least one and sometimes two reference count fields must be updated with every pointer assignment made by the LISP system. Each such adjustment requires several instructions, and pointer assignments occur on average at least once every 25 instructions [Moon84]. The processing cost of keeping reference count fields up to date is therefore somewhere between 20% and 50% of the cost of the entire LISP application.

- Some trees of nodes which are really unused may not be identified as such by reference count schemes (see below).

To guarantee that a reference count field will never overflow, the field must be as large as a pointer since every pointer in memory could conceivably point to the same node. Associating a pointer-sized field with every node in memory is generally considered to be an unacceptable memory cost. In practice [DeutB76] [WiseF77] the field is much smaller and risks overflow.

When a reference count overflows, the node associated with it must be marked as unreclaimable since it is no longer clear how many pointers reference the node. In addition, the reference count of nodes in a circular chain will always be non-zero, even if the chain is referenced by no other pointers, since every element in the circular chain is referenced by its preceeding element. Neither these chains nor nodes whose reference counts have overflowed can be reclaimed by by a reference counting scheme. For these reasons, reference count tracing schemes must be backed up by one of the previously described, more thorough tracing schemes.

No figures have been published comparing the performance of LISP systems using reference counting schemes to the performance of LISP systems using using other garbage collection schemes.

# Chapter 4

# Reclaiming Unused Storage

Where the previous section was concerned with algorithms for identifying those nodes which are in use, this section is concerned with algorithms for reclaiming all storage which a tracing algorithm has not identified as being in use. Most of these algorithms are *compacting* algorithms; that is they move all used nodes to a single contiguous region of memory, leaving a single contiguous region of unused memory. This has the advantage of making memory allocation very simple and, in a virtual memory system, of reducing the number of pages containing nodes which are still in use.

Compacting algorithms can be categorized by the relative positions of nodes which are still in use after compaction:

- In *linearizing* algorithms, nodes which are the targets of pointers in a given node tend to be adjacent to the given node after compaction.
- In *sliding* algorithms, nodes are moved to a contiguous region at one end of the address space without changing their order.

In the discussion which follows, linearizing algorithms are presented first, followed by sliding and hybrid algorithms.

Unfortunately, all of the known collection algorithms perform poorly when the number of in-use nodes is large and the number of unused nodes is small. When this is the case, the ideal garbage collector would have a cost proportional to the amount of unused memory, but this is not the case for any known LISP garbage collection algorithm. All known algorithms, with the exception of the reference count algorithm, have a cost proportional to either the number of in-use nodes or the size of memory.

## 4.1 Mark and Sweep

The oldest of reclamation algorithms is the *mark and sweep* algorithm of the first LISP interpreters [Knut73]. This technique uses any of the pointer reversal tracing algorithms to identify (mark) nodes which are still in use and then sequentially scans (sweeps) all of

memory, collecting on a linked free list any locations are not marked. The sweeping phase of this algorithm requires that the size of a node be identifiable from an examination of the node as it is encountered in the sequential scan of memory. This precludes the use of typed pointers since the sweeping phase of the garbage collector has not followed any pointer chain to nodes it encounters.

A mark and sweep garbage collector must visit every used node in memory at least once during the marking phase, and must visit every location in memory once during the sweeping phase. While this could be very expensive on a machine with virtual memory (every page of virtual memory must be paged into real memory during the sweeping phase), the cost of examining every location in memory in a real memory machine is often acceptable, given the performance characteristics of the other algorithms in this section. The memory requirements for the mark and sweep algorithm are the same as requirements of the tracing algorithm. The memory requirements for the sweeping phase are negligible when the linked list of unused memory is constructed within the unused memory itself.

The most serious drawback of the mark and sweep algorithm is memory *fragmentation.* When a contiguous unit of unused memory is smaller than the amount of memory required for a link (usually the size of a pointer), that unit of unused memory cannot be added to the free list and so cannot be re-used. Worse, even though there may be enough unused storage in total on the free list to satisfy a request for a node of a particular size, there may be no single entry on the free list large enough to satisfy the request. The only way to guarantee eliminating the fragmentation problem given variable-sized nodes is to use one of the following compacting algorithms.

## 4.2 Reclaiming Storage Using Reference Counts

Reference counts easily identify unused nodes, but reclaiming such nodes efficiently is more difficult. The problem is that when the reference count on a node reaches zero, the entire subtree rooted at that node is potentially reclaimable. The processing cost of reclaiming the tree is an issue. Ideally this cost should be distributed among the nodes in the tree as they are re-used. A more pressing concern is the memory requirement for traversing the tree.

Any naive scheme for traversing the tree and freeing nodes in it would have storage requirements similar to those of the tree tracing algorithms presented earlier. This storage can be eliminated entirely if there is a way to distinguish pointers from small integers in the first pointer field of a node (ie: use some bit which is otherwise meaningless in an unused node - perhaps the mark bit). Using the pointer reversal mechanism presented earlier and storing the current pointer location in the first pointer field of the node eliminates the trace stack. As the garbage collector works its way back to the root of the tree after visiting its branches, it examines the first pointer field of the node. If this field contains a pointer, it is the reversed pointer and all of the other pointers in the node have yet to be chased. If the first pointer field contains an integer, this integer specifies which of the remaining pointer fields is reversed. Note that this mechanism cannot be used when tracing, since it destroys the first pointer field in the node. The contents of this pointer field is irrelevant when reclaiming

storage though, since it is destroyed after what it points to has been reclaimed.

When all of the nodes in a tree are of the same size, processing costs of reclaiming the tree can be amortized among the re-allocation of the reclaimed nodes by using Weizbaum's trick [Weiz63]. When a tree of nodes is identified as no longer in use, the pointer reversal mechanism above is used to immediately reclaim a leaf node in the tree. A leaf node of course, is a node which either contains no pointers or contains only pointers to nodes which are still in use. In the latter case, the reference counts of these nodes must be decremented before reclaiming the leaf node. The reclaimed node, call it a free list *link node*, is used to link the tree being reclaimed into the free list. The node in the tree which is the target of the link node is not the root of the tree, but the current position in the tree of the pointer reversal reclamation process. Every time a new free node is required from this tree, the pointer reversal reclaimer is applied to reclaim a single node.

The most serious drawbacks of the reference count scheme for storage reclamation are memory fragmentation and the processing costs of keeping reference counts up to date. The biggest advantage to the scheme is that there are rarely large delays introduced into the LISP system by garbage collection. The only time garbage collection is necessary is when free space has been exhausted and some other garbage collection scheme is invoked to reclaim those unused nodes which the reference counting scheme missed.

## 4.3 Compacting By Reserving Forwarding Pointers

Probably the simplest compacting storage reclamation mechanism is the LISP 2 algorithm described in [Knut73] pp: 602-603. This algorithm reserves a pointer field in every node to hold a forwarding pointer. The algorithm consists of a tracing pass, a pointer adjustment pass and a final copying pass. The tracing pass can use pointer reversal, and can use the reserved pointer field to eliminate the need for a trace stack. A node is marked as being in use when its reserved pointer field is non-nil. The pointer adjustment pass scans all of memory and replaces all pointers in all in-use nodes with the forwarding pointer assigned to the node the in-use node's pointer points at. The final copying pass scans all of memory again and copies all in-use nodes to the locations their forwarding pointers point at.

The big advantage of this algorithm is its simplicity, speed [CoheN83], and the fact that it is a linearizing compaction algorithm (the significance of this will be apparent in the chapter on CDR-encoding). This algorithm is little used in modern LISP systems though, because of the requirement to reserve a pointer field in every node. The cost of this field is non-trivial in most LISP systems, since the majority of nodes in LISP systems are very small [ClarG77].

## 4.4 Two Space Compacting

The tracing by moving algorithms of the last chapter can be adapated for use as complete garbage collectors [Bake78a] [Stee75]. Memory for the LISP interpreter is split into two spaces, usually called *old-space* and *new-space*. Initially, all used nodes are contained in old-

space and all new nodes are allocated in old-space. When the LISP application requests a new node whose size is larger than the amount of space remaining in old-space, the garbage collector is activated.

The garbage collection consists of a single tracing phase, with all of the traced nodes being moved to new-space. When the tracing phase is complete, the meaning of old-space and new-space are reversed: used nodes are located in new-space and new nodes are allocated there also. When there is no more room in new-space for a request for more memory, the tree of used nodes in new-space is moved back to old-space.

The big advantage of this algorithm is that it compacts memory and that it does it relatively quickly. Each used node in the tree is visited no more than twice: once to copy the node to new-space and once to adjust the pointers in the node. The unused portions of old-space are not visited at all, resulting in a small CPU savings during garbage collection and a potentially tremendous real time savings in reduced paging in a virtual memory machine. One last advantage which is apparent in the chapter on CDR-encoding is that the collector is a linearizing collector.

The big disadvantage with this algorithm is that it requires that the garbage collector reserve memory enough to contain all of the used nodes in the LISP system (ie: new-space must be reserved). This means that the LISP system can never use more data than one half of the memory in a real memory machine. It also means that garbage collection must be carried out more frequently than in mark and sweep, since the LISP system has less unused memory to consume between collections. The partially compacting hybrid scheme presented later in this section addresses, to some extent, these limitations.

## 4.5 Pointer Munging Revisited

Morris [Morr78] invented and Jonkers [Jonk79] subsequently optimized an ingenious algorithm similar to the pointer reversing tracing algorithm. The algorithm relies on the fact that under certain circumstances, a set of locations $A_1, A_1 ... A_n$ all containing pointers pointing to a location $X$ containing $Y$ contains the same information as the chain rooted at X where:

$$\begin{gathered} contents(X) -> A_1, \\ for\ all\ i < n,\ contents(A_i) -> A_{i+1}, \text{ and} \\ contents(A_n) = Y \end{gathered}$$

Morris and Jonkers use this fact to do pointer adjustment without using forwarding pointers. This algorithm therefore requires no additional storage, either explicitly to hold forwarding pointers or implicitly to reserve large fractions of the address space for a list copying process.

Jonkers algorithm has three phases:

- a pointer reversal tracing phase to identify nodes which are still in use,

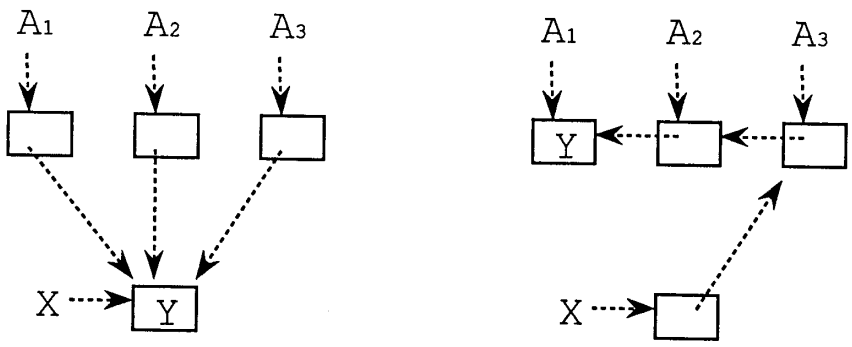



Figure 4.1: Morris's Chaining Concept

- a *forward* scan of all memory, from lowest to highest addresses, to carry out pointer adjustment of *forward pointers* (pointers which point to addresses higher in memory than the pointer's own address), and

- another forward scan of memory to carry out pointer adjustment of *backward pointers* and to copy all in-use nodes to a contiguous region of storage starting at the lowest address in memory.

The second and third passes of Jonkers' algorithm assume that all nodes contain a pointer-sized field which does not contain a pointer. Morris' original algorithm makes no such assumption, but requires that the third phase scan *backward* through memory. Scanning backwards requires that blocks of in-use nodes be searched for the beginning of the next node in the backwards direction. The cost of this searching is non-trivial [CoheN83].

The second phase of Jonkers' algorithm carries out pointer adjustment of forward pointers and converts backwards pointers into a pointer chain rooted at the mutual target of the backwards pointers. The second phase does not copy nodes, but keeps track of where the third phase will copy the nodes and uses this information to carry out pointer adjustment. To do this, the second phase scans all of memory and processes every pointer $P$ in every used node thus:

- When $P$ points to some node other that the node $M$ which contains $P$, $P$ is replaced with the contents of the non-pointer value in the pointer-sized field $A$ in the node $N$ which $P$ points at. $A$ is then replaced with the address of $P$. above. When multiple pointers point at $N$ and this procedure is applied to all of them, the result is that $A$ points to a chain of pointers ending in the original contents of $A$.

- When $P$ points to its containing node $M$, $P$ is replaced with the address of where $M$ will be copied on the third phase.

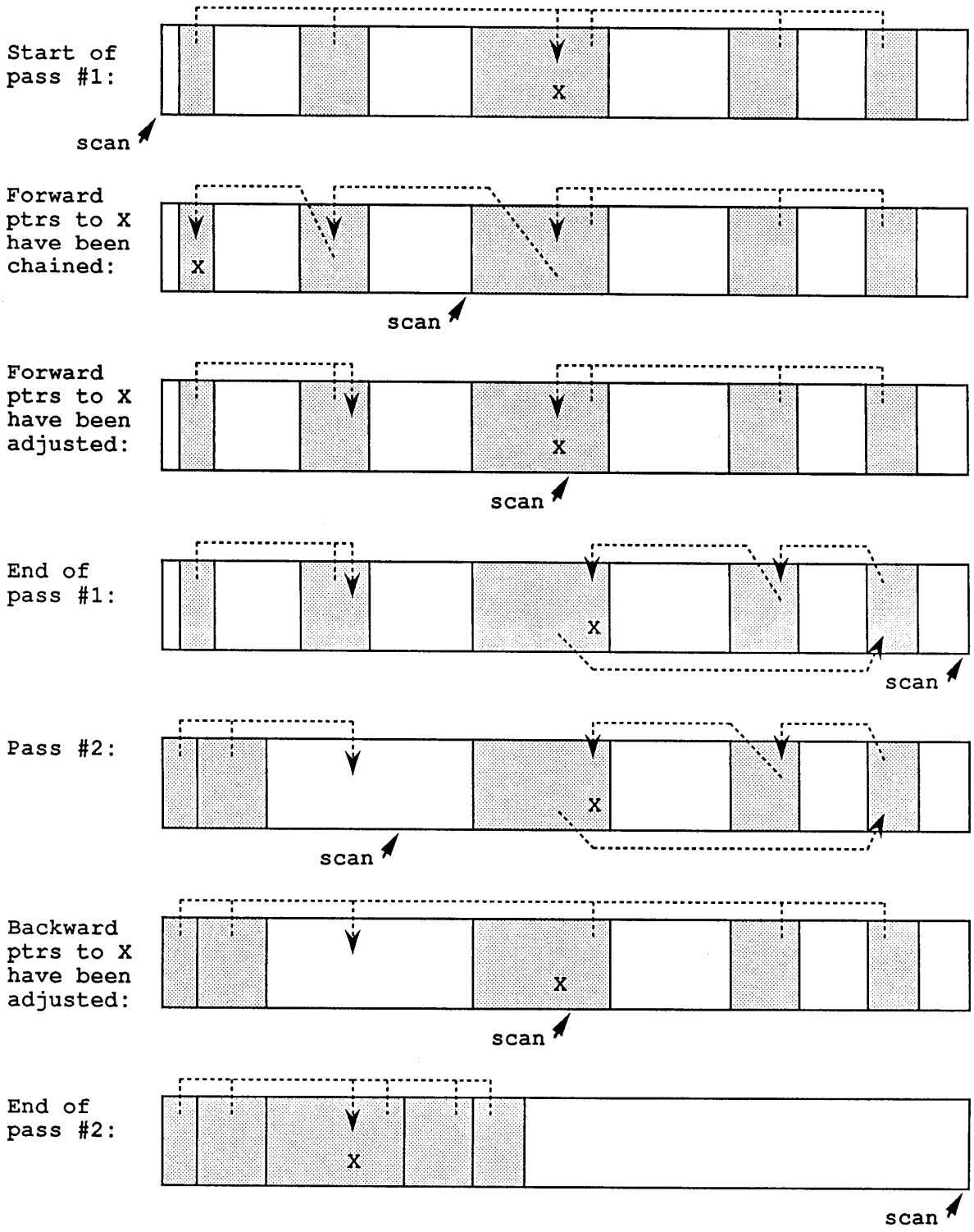



Figure 4.2: Jonker's Algorithm

If this were all that the second phase accomplished, the end of the second phase would see every used node chained to all of the pointer fields that reference it. In fact, the second phase, upon encountering a used node, carries out pointer adjustment on all forward pointers to that node. It does this by chasing the chain of pointers rooted at the node and replacing each of them with the address the node will be copied to in the third phase. The contents of the pointer-sized field in the node is restored from the last field in the pointer chain. This pointer adjustment cannot be carried out in the third phase because, by the time the third phase encounters a node, it will already have moved all of the nodes containing forward pointers to that node.

The third phase scans all of memory and copies backwards to the beginning of memory all used nodes. After the copy, it carries out pointer adjustment of all backward pointers to the node in the same way that the second phase carried out pointer adjustment of forward pointers.

Both the second and third phases, upon encountering a node, must be able to determine whether or not a chain of pointers requiring adjustment are rooted in the pointer-sized field. Jonkers suggests that by using an appropriate structure for information within the node, this determination can be made simply by examining the pointer-sized field. However, in general, it seems that each node must have a bit reserved in it somewhere to determine whether the pointer-sized field contains its original contents or not. If a priming sweep is made through memory, coalescing all of the unused nodes and identifying in them the lengths of the used regions between the unused regions, the mark bit used by the pointer reversal tracing algorithm can be re-used to indicate that the pointer-sized field in the node contains its original contents.

Both the second and third phases, upon chasing a pointer chain to carry out pointer adjustment must be able to recognize the end of the chain. Again, Jonkers suggests that no additional storage is required for this, but it seems that in general, a bit in each pointer in the chain must be reserved to determines whether or not the target of the pointer is the end of the chain. Jonkers' algorithm is described in detail in Appendix E.

Jonkers' algorithm performs almost as well [CoheN83] as the LISP 2 algorithm which reserves forwarding pointers and requires somewhat less storage than that algorithm. In Jonkers' algorithm, the reserved word for forwarding pointers can contain any information except a pointer to another node.

## 4.6 Table Collectors

A relatively large body of literature has been published describing a class of algorithms which Cohen describes as *table collectors* [CoheN83] [Cohe81]. All of these algorithms involve at least three phases:

- A marking phase identifies all used and unused memory.
- A table describing used memory is constructed, to be used when adjusting pointers in the next phase. This table usually consists of pairs of addresses: the start address of

a block of used nodes and the new address that block will occupy after compaction. These tables can be simple linear chains of pairs of addresses [FitcN78] [HaddW67] [Wegb72a] [HaddW67], hash tables [FitcN78], or binary trees [TeraG78].

- The final phase searches the above constructed table to adjust each pointer in all used blocks of memory and then copies those blocks to their new locations in memory.

The simplest, least processor efficient tables are guaranteed to fit into unused storage, provided that the smallest unused node is large enough to contain two pointers and a flag [HaddW67]. Other tables, while more efficient than simple chains of address pairs, require that an additional fixed amount of storage be allocated for the garbage collector. These table driven collectors, on the surface, seem less efficient than other compacting collectors because of the cost of searching the table during pointer adjustment. In theory, they should cost about $O(N\ log(N))$ where $N$ is the number of in-use nodes. In practice [CoheN83], these algorithms appear to perform almost as well as the pointer munging algorithms described above.

## 4.7 Hybrid Algorithms

A recently published algorithm by Lang and Dupont [LangD87] combines mark and sweep with two space compacting. This algorithm divides memory into three areas: ms-space (mark and sweep space), old-space and new-space. Ms-space is a region of memory collected with a mark and sweep algorithm, old-space is a region of memory collected with a two-space compacting algorithm and new-space is a region at least as large as old-space which is reserved for use in the two-space compacting process. After every garbage collection, old-space is entirely unused and is designated as the new new-space. New-space, after every collection, is partly filled with nodes from the old old-space and is designated to be part of ms-space. The most fragmented region in the new ms-space is then designated to be the new old-space for the next collection.

This algorithm compacts only the most fragmented subset of memory with every garbage collection. It represents a trade-off between the space-efficiency of a mark and sweep algorithm (which requires very little reserved memory) and the time efficiency of the two-space compacting algorithm (which visits only used nodes, no more than twice each). The hybrid algorithm reserves only a small region of memory when memory is scarce, rather than reserving one half of memory like the pure two-space compacting algorithm. When memory is plentiful, the algorithm eliminates ms-space altogether and behaves like the processor efficient, pure two-space compacting algorithm.

This adaptation occurs dynamically in the hybrid algorithm. When memory becomes scarce, the algorithm simply reduces the size of old-space. When old-space becomes new-space following the next garbage collection, new-space will be similarly reduced in size, making more memory available for use by the LISP system. When memory becomes plentiful, new-space can be increased in size whenever it is adjacent to an unused area in ms-space. Such adjacency can be forced by choosing the largest unused area in ms-space following a

garbage collection, removing that area from the free list, and designating old-space to begin adjacent to this area. Following the next collection, new-space (the old old-space) will be adjacent to this large, unused area and can therefore be expanded.

If there is enough memory, and if multiple regions of ms-space are highly fragmented, several old-space and new-space regions can be assigned. In practice, it seems unlikely that multiple regions could be justified. Multiple, small, two-space compaction efforts will not yield the same reduction of fragmentation as a single compaction effort, and multiple efforts will not linearize memory to the same degree as a single effort. Nothing has yet been published describing effective algorithms for choosing which region of memory should be the next old-space or for determining an optimal size for old or new-space. A simple implementation of the hybrid algorithm can be seen in Appendix F.

An as yet unpublished hybrid algorithm invented by the author was inspired by Lang and Dupont's efforts. This algorithm makes three passes through memory:

- The first pass is a straightforward mark and sweep pass to identify unused areas in memory.
- The second pass is a tracing pass similar to that of Lang and Dupont's where nodes in ms-space are marked and nodes in old-space are copied to new-space. In this case though, new-space is the unused memory identified by the first pass.
- The third pass copies the contents of unused memory (new-space) back into old-space in consecutive locations in memory. As this information is copied, the free list is rebuilt.

The second pass is able to employ unused memory as new-space because Cheney's trace-by-moving algorithm accesses new-space sequentially. Sequential access to the free list can be arranged simply by keeping track of the current free item and the current position in that item. Not all of the space in an unused region of memory can be used as new-space. A size field must be reserved in each unused region to determine where the next used region begins. Information describing either the location of the next unused region or the size of the in-use region following each unused item is desirable as well. If this linking information is not present, in-use regions must be sequentially scanned in pass three, increasing the processing cost of the algorithm somewhat. This algorithm is illustrated in Appendix G.

The differences between the algorithms can be summarized thus:

- The memory cost of Ginter's algorithm is equal to the memory cost of a mark and sweep algorithm. The memory cost of Lang's algorithm is somewhere between that of a mark and sweep algorithm and that of the more memory-expensive trace-by-moving algorithm.
- The processing cost of Ginter's algorithm is the cost of a mark and sweep pass of all memory, a trace-by-moving pass of all memory and a copying pass of new-space (which may contain only a subset of all of the in-use nodes). The processing cost of Lang's algorithm is somewhere between that of a trace-by-moving algorithm and the more expensive mark-and-sweep algorithm.

- The degree of compaction of Ginter's algorithm is generally greater than that of Lang's algorithm. The exception to this rule is when memory is so badly fragmented that most unused regions contain barely enough room for the size and link information.

- Ginter's algorithm requires strictly fewer garbage collections than Lang's when Lang's new-space is of non-zero size. Ginter's algorithm reserves no memory from the LISP system, and so provides the LISP system with more memory to consume than Lang's algorithm.

- Lang's algorithm can be executed in parallel with the LISP system while Ginter's cannot. Parallel execution requires random access to new-space which is impossible in Ginter's algorithm.

- Both algorithms are linearizing compaction algorithms for some subset of memory. Ginter's algorithm linearizes slightly better than Lang's since Ginter's old-space is usually larger than Lang's.

It is not clear from the above comparison which of the hybrid algorithms performs better in general. Other hybrid algorithms are possible but nothing has been published regarding them. For instance, a priming pass of a table driven compaction algorithm could slide all unused memory in a single region. A trace-by-moving compaction pass could then move a subset of the in-use nodes to that region in a linearizing fashion. A final table driven pass could slide the remainder of the in-use nodes up against the linearized region, leaving a single contiguous unused region to speed up allocation. Nothing has been written describing such hybrid algorithms.

# Chapter 5

# Compact Representation of Lists

Empirical studies of LISP systems [ClarG77] [ClarG78] show that over 50% of the data used in an average application consists of list nodes containing exactly two pointers (usually called the *car* and *cdr* pointers, respectively). In over 70% of these nodes the second pointer points to another list node. In addition, in about 25% of list nodes, the second pointer points to NIL and less than 3% of all list nodes are referenced more than once by another list node. These statistics have lead to a variety of proposals to represent list nodes more compactly. These techniques have become known as *cdr-encoding* techniques, and generally involve eliminating the second pointer in list nodes as frequently as possible.

When these techniques are 100% effective, the empirical results above indicate that at most 50% of the space in 95% of the list nodes can be eliminated. This means that the space taken up by data in a typical LISP application can be reduced by at most 23%.

## 5.1 Encoding Techniques

The earliest documented encoding technique was Hansen's [Hans69]. Hansen represented a list node as exactly two pointers, one pointing to the car of the list and the other to the cdr. Hansen attempted to *linearize* lists by eliminating the cdr from the node as frequently as possible, and by making consecutive car's occupy consecutive memory locations.

Hansen reserved a bit in each of these pointers to identify whether the pointer was really a car pointer or whether it was a cdr pointer instead. Hansen required an occasional cdr pointer to indicate the end of a list. Any cdr pointer to an atom (including NIL) indicated the end of a list and any cdr pointer to another list node indicated merely a continuation of a list. Continuations are required any time a portion of a list structure is shared between two lists, since the immediate predecessor to the shared portion can be only one of the two lists sharing it, not both.

All cdr-encoding techniques rely on the assistance of a compacting, linearizing garbage collector. While it is possible to build a cdr-encoding *CONS* operator, it is the garbage collector which periodically visits every used node in memory and so has the opportunity to carry out large scale linearization. Hansen's paper [Hans69] described in some detail a primitive

```
/* return the CDR of a list node */
NODE **cdr (list_node)
NODE **list_node;
{
     return ((NODE **) (list_node [1]));
}
```

Figure 5.1: The Original *CDR* Operator

```
/* return the CDR of a possibly compact list node */
NODE **cdr (list_node)
NODE **list_node;
{
     if (list_node [1] && REAL_CDR)
          return ((NODE **) (list_node [1]));
     else return (list_node + 1);
}
```

Figure 5.2: The Modified *CDR* Operator

linearizing garbage collector. Such linearization is a trivial by-product of many very of the simple algorithms invented since Hansen's paper was published [Chen70] [Clar76] [Knut73]

Hansen's algorithm required modifications not only to the garbage collector, but to the *CDR* and *REPLCDR* operators as well. The *CDR* operator was modified from something which simply returned the contents of the cdr pointer (figure 5.1) to something which recognized compact encodings (figure 5.2).

The biggest disadvantage of Hansen's proposal is that there is no way to reliably encode the *REPLCDR* operator. When this operator is applied to a compactly represented list node $X$, the word $Y$ following the car of $X$ must be replaced with a cdr pointer. When $Y$ already contains a cdr pointer, the replacement can be performed without loss of information. When $Y$ contains $X$'s successor node, the replacement cannot be carried out without destroying the contents of the successor node. These contents cannot even be moved to make room for the new cdr for $X$, since some other list may have saved a pointer to the old successor node.

More recent proposals [LiH86] [Bake78a] address this problem by redefining the meaning of the "cdr pointer" bit. These proposals intepret this bit as meaning "go indirect through this pointer to get what you're really looking for". Pointers with this bit set are called *indirection* pointers while pointers where this bit is clear are called *content* pointers. List structures in the new representation are encoded exactly as they are encoded in Hansen's method, but the *REPLCDR* operation is now possible.

To effect a *REPLCDR*, the car of $X$ is replaced by an indirection pointer to a new, two pointer node $N$. The car of $N$ is set to the old contents of the car of $X$ and the cdr of $X$ is set to the second argument of the *REPLCDR* operator. If $X$ is the subject of repeated *REPLCDR*

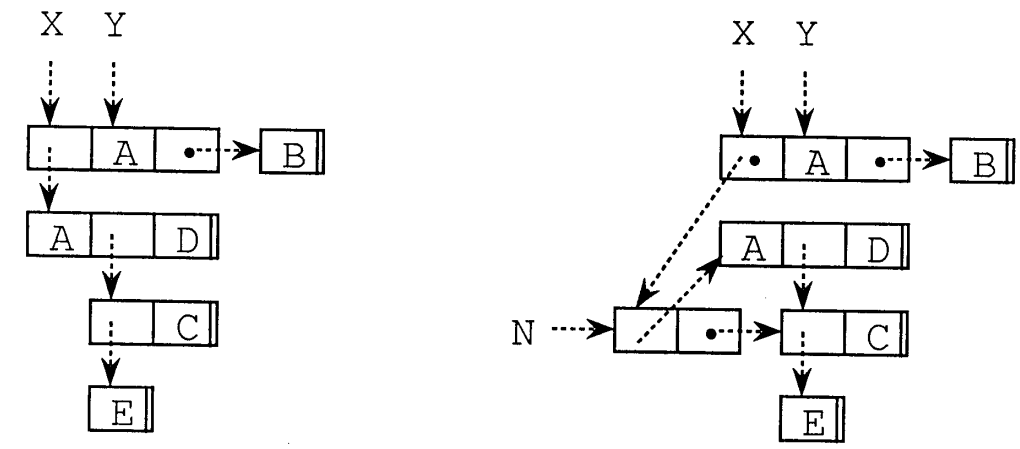



Figure 5.3: A REPLCDR Operation on a CDR-Encoded List

operations, the car of $X$ can become the root of a very long chain of indirection pointers. These chains can be eliminated if the *REPLCDR* operator chases indirection pointers in $X$'s original car pointer. If only the *CAR, CDR* and *REPLCDR* operators chase indirection pointers, then no other operators need be modified to implement cdr-encoding.

The more recent proposals all reserve two bits in the car pointer of a list node. The two bits are required to distinguish between three possibilities:

- the current pointer is an indirection pointer (and is the end of the list only if it points to an atom),
- the current pointer is a content pointer and is not the end of the list,
- the current pointer is a content pointer and is the end of the list.

The last case indicates that the cdr pointer for the list node, were it present, would be NIL. Two bits actually can encode four possibilities and in all of the documented algorithms actually do encode four possibilities [LiH86] [Bake78a]. In all cases though, the fourth possibility is either redundant or is information which could be encoded as effectively some other way.

## 5.2 A Compact *CONS* Operation

Early efforts in this field relied on the garbage collector to convert lists to a compact, linear representation. The *CONS* operator (which allocates a list node) in these early implemen-

```
/* a compact cons operator */
NODE **cons (car_stuff, cdr_stuff)
NODE *car_stuff;
NODE **cdr_stuff;
{
     NODE **new_node;              /* what cons will return */
     int pos;                      /* position in large cons cell */
     extern int bibop_type ();     /* node type using BIBOP */
     extern int cons_size;         /* global variable */


/* : check if the cons can be accomplished compactly */
     if ((bibop_type (cdr_stuff) == LIST) &&
         (*(cdr_stuff - 1) == UNUSED)) {
          new_node = cdr_stuff - 1;
          *new_node = car_stuff;
          }


/* : do the cons the hard way */
     else {
          new_node = (NODE **) malloc (cons_size * sizeof (NODE *));
          for (pos = 0; pos < (cons_size - 2); pos++)
               new_node [pos] = UNUSED;
          new_node = new_node + cons_size - 2;
          new_node [0] = car_stuff;
          new_node [1] = (NODE *) cdr_stuff;
          }
     return (new_node);
}
```

Figure 5.4: A Compact *CONS* Operator

tations returned list nodes containing two pointers and constructed lists in the same way as in non-linearizing LISPs. Li and Hudak [LiH86] describe a modification to *CONS* which constructs compact lists between garbage collections. This optimization should somewhat decrease the frequency of garbage collections by consuming memory at a slower pace during list construction.

Li and Hudak suggest that *CONS* allocate a block of $N$ pointers, use the highest addressed two pointers in the block to construct a list node $X$ and mark the remaining pointers as unused. Marking the pointers as unused can be accomplished either by using the "spare" bit pattern described above or by reserving a location in memory to indicate an unused pointer and then using the address of this reserved location as an "unused" indication. A subsequent *CONS* operation of a list node $Y$ whose cdr is $X$ can use the unused pointer field before $X$ to construct a compact representation of $Y$ (see figure 5.4).

This strategy will delay garbage collection only when it results in less waste space than

a non-compact list representation. Choosing $N$ too large will cause such waste. Choosing $N = 2$ reduces the compact *CONS* to a normal *CONS*. Li and Hudak suggest leaving the choice of $N$ to the programmer. When the default for $N$ is 2, the application performs as well as it would in the abscence of a compacting *CONS*. When the programmer knows, either statically or dynamically, the exact length of a list being constructed, the programmer can instruct *CONS* to use a vector of the appropriate length and achieve the maximum benefit possible from a compacting *CONS*.

# Chapter 6

# Generation Scavenging

Generation scavenging [Unga84] [LiebH83] [Cour88] [Moon84] is an algorithm intended to reduce the frequency of garbage collection by collecting only those regions of memory which are likely to contain unused nodes. The algorithm assumes that:

- recently created nodes become unused much more quickly than do nodes which have been in-use for some time, and
- the number of pointers from old nodes to young nodes is relatively small.

While no empirical studies yet published verify these assumptions, generation scavenging has improved the performance of several systems in which it was implemented [Moon84] [Unga84] [Cour88].

Generation scavenging divides memory into two or more regions, each containing nodes of a certain age range. New nodes are allocated in the region containing the youngest nodes unless the application has instructed the LISP system that a number of nodes are to be allocated in another region. The latter functionality allows application developers to make the best use of the algorithm when it is possible to predict how long those nodes are likely to be in-use.

Every node in the system has an associated age. This is usually encoded in a bit field somewhere within the node. A single bit will suffice, if storage within nodes is scarce. When a region of memory fills, it, and all younger regions, are garbage collected. All of the nodes in each of these regions which are older than the oldest age allowed for the region are copied to a region of memory containing older nodes.

The trickiest part of generation scavenging is the fact that older nodes in one region of memory may contain pointers to younger nodes in another region of memory. If a younger region of memory is garbage collected, the trees of nodes rooted in all older regions of memory must be marked as in-use. On the surface, this appears to require that all in-use nodes in all of memory be traced whenever any region of memory is garbage collected. If this were the case, the generation scavenging algorithm would offer little hope of a performance improvement.

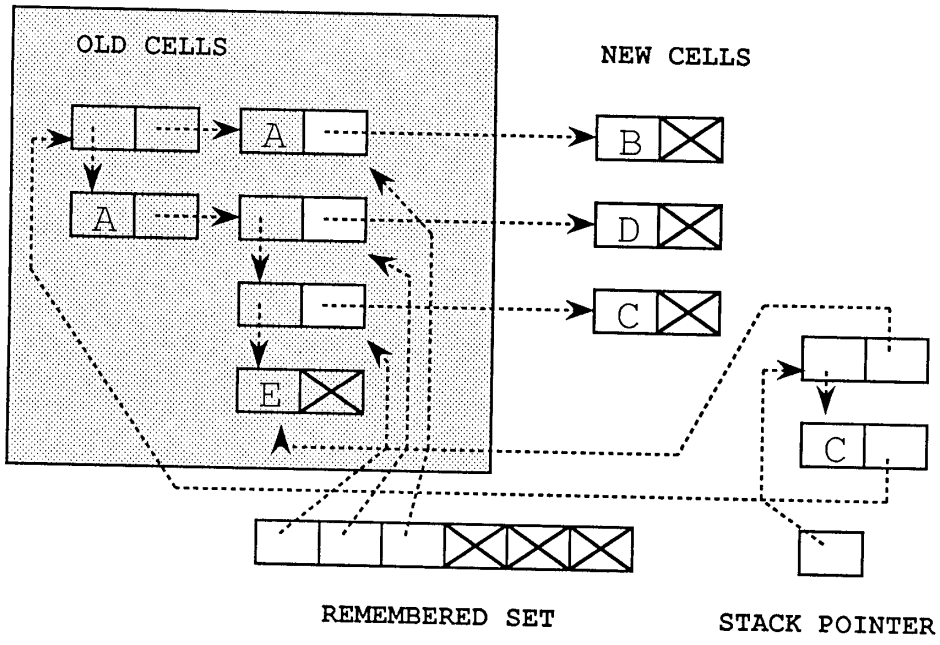



Figure 6.1: Ungar's Generation Scavenging

The algorithm gets around this requirement by maintaining $(N^2-N)/2$ tables of pointers from older regions of memory into younger regions of memory. Every pair of older/younger regions has an associated table of pointers to locations in the older region which point to locations in the younger region. Ungar calls these tables the *remembered sets*. Every garbage collection of a region of memory $A$ must mark as in-use those trees of nodes identified by each of the remembered sets $R$ corresponding to each pair of spaces $A/B$ where $B$ is younger than $A$.

Every garbage collection of a region of memory $A$ also updates all of the remembered sets associated with $A$:

- Remembered sets associated with $A$ and an older region $B$ are pruned of all references to locations in $B$ which no longer point to $A$.
- Remembered sets associated with $A$ and a younger region $B$ are discarded and are completely rebuilt during the tracing of in-use nodes in $A$. Whenever a pointer is found which points to a location in $B$, the address of that pointer is added to the new $A/B$ remembered set.

In addition, it would appear that every set instruction to any location in any region $A$ must be examined to see if what is being set is a pointer to a location in a younger region $B$. When such sets occur, the address of the pointer must be added to the $A/B$ remembered set.

In practice, it is possible to eliminate these checks for sets to locations which are known to reside in the youngest region of memory. These checks can therefore be eliminated from set instructions in many primitive LISP functions, and from set instructions to local variables in the stack if the stack is restricted to lie entirely within the youngest memory region.

All published implementations of generation scavenging in LISP accomplish checking of set instructions in hardware. No published results indicate whether or not generation scavenging results in improved overall performance when checking set instructions is done in software. Moon [Moon84] observed that about one instruction in 25 on a LISP machine is a set instruction and speculated that a software implementation would result in poorer performance by between a factor of 1.1 and 2.

Ungar [Unga84] implemented generation scavenging in software for Smalltalk. His results show that one instruction in 23 is a set instruction, but in Ungar's implementation those instructions were interpreted. Each interpreted instruction required approximately 25 hardware instructions, resulting in a set instruction about once every 600 hardware instructions. Ungar's results shows that generation scavenging improved the performance of the Smalltalk system. This is because the 5–10 instruction check of set instructions degraded the Smalltalk system's performance by only $10/600 = 0.2\%$. A similar implementation in LISP would degrade the LISP system performance by $10/25 = 40\%$.

# Chapter 7

# Variable Binding Strategies

Variable binding strategies are worth examining in the context of memory management, since this activity is responsible for generating most of the "garbage" collected by the garbage collectors discussed so far. LISP implementations using inefficient variable binding strategies can spend up to 80% of their time running the garbage collector [HollSSB80].

## 7.1 Variable Binding and Closures

Variable binding strategies allow LISP functions to read and write values associated with LISP variables or *symbols*. In most implementations, every LISP symbol has an entry in a global symbol table, and this entry contains things like the name of the symbol, its property list and its *value cell*. The value cell of a symbol contains a pointer to the location holding the *top-level* value of the symbol.

Any LISP procedure can associate a local value with a symbol. Such a local value is called a *lambda binding* and is *active* only until the procedure which defined it exits. A lambda-binding *environment* is the set of lambda bindings which are active at a particular point in time. An environment can be *captured* and associated with a LISP procedure. Such an association is called a *closure*. A closure is invoked by temporarily inactivating all lambda bindings in the current environment (by *suspending* the current environment), reactivating all of the bindings in the environment in the closure and then invoking the procedure in the closure.

In all LISP implementations, a reference from a LISP procedure to a lambda-bound symbol affects the most recent active lambda-binding of the symbol. If there is no active lambda-binding for a symbol, references to the symbol affect the top-level value of the symbol. If a lambda-binding has been captured in several closures, modifying the binding in one closure modifies the binding in all of the closures.

```
; apply procedure on each element of list-1 until
; the first non-nil result
(defun or-map (procedure a-list)
     (if a-list
         (or (procedure (car a-list))
             (or-map procedure (cdr a-list)))))

; Return the first element of list-2 which is an
; element of list-1
(defun find-first-member (list-1 list-2)
     (or-map (lambda (element)
                  (if (member element list-1)
                      element))
             list-2))
```

Figure 7.1: The Downward Funarg Problem

## 7.2 The Funarg Problem

The problem of symbol references in procedures which are passed or returned as functional arguments is called the *funarg problem.* The problem with these references is that they can yield unexpected results. For example, in the code fragment in figure 7.1, it is clear that when the lambda function is executed, the programmer intends the *list-1* symbol to refer to the *list-1* lambda variable in the first line of *find-first-member.* In fact, when that procedure is executed by *or-map*, the most recent lambda-binding for *list-1* will be to some subset of *find-first-member*'s *list-2* variable. This is known as the *downward* funarg problem because functions are being passed as arguments to procedures *down* the calling subtree of a program.

Figure 7.2 illustrates the *upward* funarg problem. In this case, a function is being passed *up* the calling subtree of a program. It is clear that the programmer intended the $n$ in the lambda function to refer to the $n$ passed into *make-nums.* In fact, when *s-cdr* is evaluated, that $n$ will refer to whatever the top-level value of $n$ is (if there is one).

The solution to the funarg problem is to pass closures, not procedures, as function arguments. If the *lambda* function had captured its environment in both of the above cases, then when those closures were invoked, the symbol references in the closures would have been resolved in the expected manner. In *lexically scoped* LISPs, this is the case. In *dynamically scoped* LISPs, the built-in procedure *function* is usually responsible for creating closures. In a dynamically scoped LISP, all of the *(lambda ...*'s of figure 7.1 and 7.2 would have to be replaced by *(function (lambda ...* for the examples to work correctly.

## 7.3 Tree Structured Deep Binding

A simple representation for lambda-bindings is an association list [Bake78b]. Each element in the list is a *(symbol . value)* pair and the *car* of the list is the most recent lambda-binding.

```
; Return a stream of consecutive integers starting from N.
; (A stream is a list whose CAR is the next value in the
; stream and whose CDR is a procedure to call to create the
; following number/procedure pair.
(defun make-nums (n)
     (cons n
           (lambda (prog)
               (setq n (+ 1 n))
               (cons n prog))))

; Return the first element in a stream
(defun s-car (stream) (car stream))

; Return the rest of a stream
(defun s-cdr (stream)
     (apply (cdr stream)
            (list (cdr stream))))

; print the first two elements in a stream
(setq num-stream (make-nums 25))
(s-car num-stream)                ;should be 25
(s-car (s-cdr num-stream))        ;should be 26
(s-car (s-cdr (s-cdr num-stream))) ;should be 27
```

Figure 7.2: The Upward Funarg Problem

If a symbol does not have an associated value in the list, its value is the top-level value of the symbol. This mechanism is called *deep binding.* The environment list in a deep binding LISP can be thought of as a tree whose root is the last element in the list. A closure in a deep bound implementation is simply a pointer to the first element of the environment list. Any closure which captures the environment captures a path from a leaf (a set of bindings corresponding to some procedure) to the root. Since deep binding allows closures, it can accommodate both lexically and dynamically scoped LISPs.

Invoking a function in a deep bound implementation means *cons*'ing a symbol/value pair to the front of the environment list for every lambda variable in the function. The cost of invoking a function is proportional to the number of lambda variables in the function. Returning from a function and restoring the original environment is accomplished by removing the function's lambda-bindings from the environment. In most implementations this is accomplished by a single pointer assignment. Evaluating a symbol in a deep bound implementation means searching the environment list for a value for that symbol. The length of the environment list in a deep bound implementation is potentially unbounded, because it grows with each function invocation. This means that evaluating a symbol in a deep bound environment involves a search whose length can grow as large as large as all of memory.

The memory cost of such an environment list is substantial. The tree must be maintained as a list in dynamically allocated storage because a closure can capture any path from leaf to root in the tree. Closures, however, tend to occur infrequently in the course of a LISP application, which means that most of the storage allocated to an environment list becomes garbage very quickly, making frequent garbage collection necessary. More memory efficient environment implementations are considered later in this section.

Compiled LISP implementations can somewhat reduce the memory and processing costs of a deep bound implementation. Compiled implementations can use a stack for *local* symbols - symbols which are not referenced by any code deeper in the calling tree. In a dynamically scoped implementation, user intervention in the form of *declare* statements [Moon78] is necessary to identify local variables, since the compiler cannot identify those variables which will be referenced by code deep in a calling subtree. When local variables are relegated to a stack, fewer variables appear in the environment list, which means both searching and maintaining the list is cheaper.

In a lexically scoped implementation, the compiler requires no user intervention to identify local variables. In a lexically scoped environment, the environment in which a function is defined is the environment in which it will be executed. If a function defines no further functions itself, then all of the variables in that function are local. When a function does define nested functions, only non-local references from those nested functions to variables in the defining function need be placed on the environment list [BartJ86].

In a compiled, lexically scoped environment a further optimization is possible. A lexically scoped compiler can identify at compile time, all of the references to a particular environment binding. Thus, instead of storing the environment list as a list of bindings, the environment list can be stored as a list of records, with lambda-bound values stored consecutively in the record [Bake78b]. A compiled reference to one of these values requires no searching. Such a

reference involves generating code to access a known offset in an environment record which is a known number of records from the beginning of the list.

## 7.4 Tree Structured Shallow Binding

Deep binding keeps lambda-bindings in an environment and searches that environment for values. *Shallow binding* keeps lambda-bindings in the top-level value cells of symbols and keeps old values of these symbols in the environment. A shallow bound function call involves saving all of the values of top-level symbols which are about to be lambda-bound in the environment and then modifying the top-level values of these symbols to their new lambda-bindings. The advantage to this is that a function referencing a symbol does not need to search an environment for the symbol's value - the value is immediately available in the symbol's value cell.

Baker [Bake78b] describes a general method which can be used to convert a deep bound environment to a shallow bound one. Baker's environment list contains:

$$((symbol_1.value)...(symbol_N.value)())$$

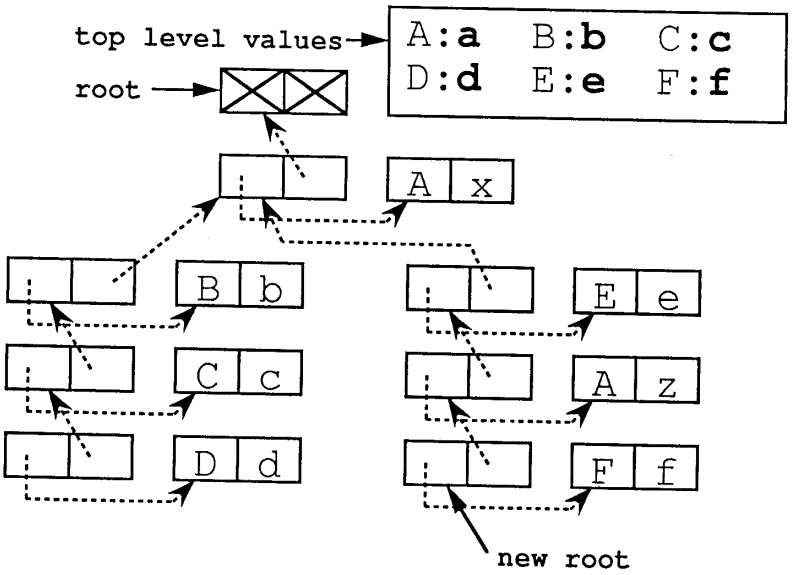
The trailing *()* is a place holder indicating the binding associated with the root of the environment tree. Baker converts this deep bound environment list to a shallow bound one by the application of the *reroot* procedure in Appendix H. This procedure

- reverses the environment list using destructive list operations,
- traverses the reversed list and exchanges values in the list with top-level values and finally
- rotates the list by one element during the traversal.

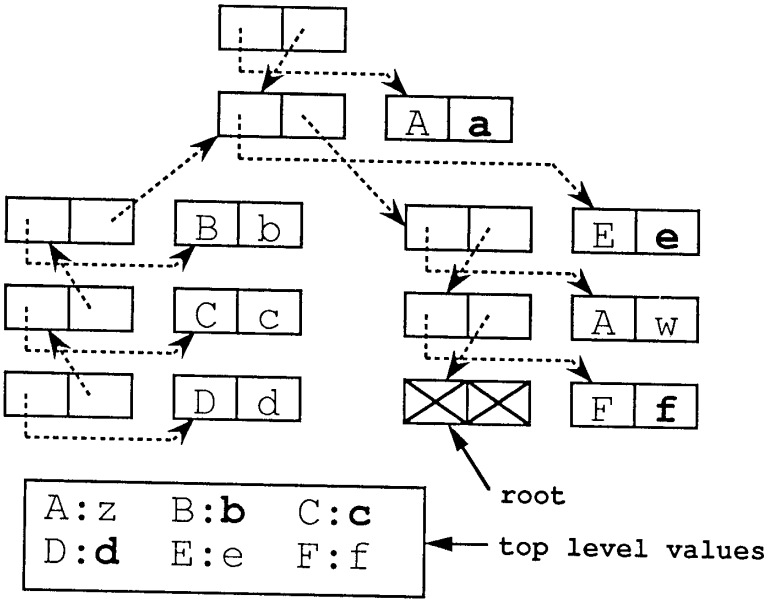
Shallow binding is achieved during the exchange of values between the environment list and the top-level symbol values. The list reversal is necessary because bindings must be exchanged with top-level values in the reverse of the order in which they appear in the environment list. If a shallow bound reference to a symbol value cell is to return the same value as a deep bound search, the value which the deep-bound search would have found first must be placed into the value cell last.

The list is left reversed and is rotated to allow closures to work properly. Any closure which captured all or a subset of the environment list will refer to the modified environment list after the application of the algorithm. A closure which captured the entire environment list would have captured an environment pointer to the head of the list. Since Baker's algorithm converts the head of the list into its tail, a deep binding search of the environment list will discover that it contains no bindings and will conclude that the current value for any symbol is contained in that symbol's value cell.

A closure which captured some subset of the list will find that its captured pointer refers to an element of the modified list. The reversal and rotation guarantees that the captured




a) The binding tree before re-rooting




b) The binding tree after re-rooting

Figure 7.3: Baker's Re-rooting Algorithm

pointer points to a deep binding environment describing the difference between the current set of top-level values and the lambda-bindings captured by the closure. Baker's algorithm can be applied again to this deep bound environment to convert it to a shallow bound environment for the evaluation of the function in the closure.

Baker calls his algorithm a rerooting algorithm, since it effectively moves the root of the environment tree out to a leaf. Baker's article [Bake78b] gives an inductive proof that his algorithm preserves lambda bound values for both functions and closures. Baker further points out that since his algorithm can be applied to an environment list at will, it can be used to give the programmer control over whether deep or shallow binding should be carried out during the evaluation of a function.

It is also worth noting that when shallow binding is carried out consistently using Baker's algorithm in a dynamically scoped environment, the cost of shallow binding a function is proportional to the number of lambda variables in the function. The cost of applying shallow binding to a closure however, is proportional to the length of the environment list in the closure. This makes invoking a closure in a shallow bound implementation as expensive as evaluating a symbol in a deep bound implementation.

## 7.5 MACLISP Shallow Binding

Maclisp [Moon78] implemented shallow binding using a stack in an attempt to reduce the memory and processing costs of lambda binding. The processing cost of invoking functions and closures in a stack based environment is identical to the cost of such invocations in a tree based environment. The memory cost is much lower though, since the memory used for lambda-binding for a function can be immediately reclaimed when the function exits.

The Maclisp mechanism, while more memory efficient than a tree based environment implementation, is incapable of dealing with the upward funarg problem. Downward funargs in Maclisp are implemented as closures where the environment pointer in the closure is a pointer to the stack frame which defined the function argument. Upward funargs can use no such mechansim since the target of the stack frame pointer in the closure would cease to exist as soon as the closure is returned.

## 7.6 Phantom and Spaghetti Stacks

A variety of stack-based strategies have been proposed to reduce the memory cost of binding while providing a closure mechanism powerful enough to address the funarg problem. Each of these strategies stores environment information in a stack rather than in a garbage collected store *most of the time*. Such strategies have been deemed absolutely essential in implementing SCHEME [HollSSB80], since an implementation entirely within garbage-collected memory was found to consume 80% of its time in garbage collections.

McDermott [McDe80] proposed a simple, if slightly inefficient, mechanism for implementing lambda-binding environments in a stack. McDermott's idea was to store the tree

structure of such environments on a stack as long as no part of the environment was captured. As soon as some binding in the stack was captured, the entire stack below the binding was copied into garbage-collected memory with forwarding pointers to the new copy left in the stack. The forwarding pointers guaranteed that the captured environment was correctly shared when multiply captured. The biggest drawback of this method was the cost of copying a potentially large stack from one place to another.

Stallman [Stal80] quickly addressed this drawback in proposing *phantom stacks*. Stallman suggested that stacks be allocated in garbage-collected memory to begin with, so that when an environment was captured, no copying would be necessary. The captured portion of the current stack could simply be abandoned to the garbage collector. When one of these stacks overflows, it is also abandoned, and a new stack is allocated.

The biggest problem with phantom stacks lies in determining what size of stack to use when allocating a new stack in garbage collected memory. If stacks are allocated too small, they will overflow frequently and be frequently abandoned. When allocated, stacks should be large enough so that most of stack abandonment is a result of stack capture activity rather than stack overflow.

Another scheme to use stacks to hold environment information is the *spaghetti stack* effort of Bobrow and Wegbreit in 1973 [BobrW73]. This algorithm involves copying captured stack frames to garbage collected store when those frames would normally be popped. A complex reference count scheme is used to determine whether or not a frame on the stack has in fact been captured. Because of its complexity, this technique is rarely used in practice [Stal80] [Danv87].

A recent optimization of this technique [Danv87] involves allocating environment frames in garbage-collected memory and then reclaiming uncaptured frames when the procedure which created them returns. Frames are marked as captured by setting a bit in the frame when it is captured, and by manipulating a global variable *CAPTURE*. The capture bit in the frame indicates that the frame is the head of a chain of captured frames. The *CAPTURE* variable indicates that the current frame, marked or not, is in fact a captured frame. A frame can be reclaimed only when both indicators are clear. The *CAPTURE* variable must be set whenever control is transferred to a frame marked as captured.

A final optimization which can be applied to any of these stacking strategies is the detection of downward funarg/closures. Creating a closure and passing it to another procedure as an argument can be done safely in a stack-based environment since the environment captured by the closure is still present on the stack when the function argument is invoked. All of the stack-based environment algorithms recommend examining the arguments of the *rplcar, rplcdr, setq* operators and the (often invisible) *return* operator. When a closure is detected as an argument to one of these operators, that closure risks use as an upward funarg. Only when the risk of such use is detected, should any of the stack based algorithms register a capture of a portion of the stack.

No performance measures are available for any of the techniques in this section.

# Chapter 8

# Continuations

A *continuation* is a closure which has captured not only a lambda-binding environment, but a control environment as well. When a continuation is invoked, the procedure in the continuation is invoked at the location in which the continuation was created. When the procedure in a continuation exits, control returns to the procedure that created the continuation. In contrast, when a closure is invoked, the procedure in the closure is invoked at its beginning. When the procedure in a closure exits, control returns to the procedure that invoked the closure.

Continuations are therefore comparable more to coroutines than to subroutines. Continuations are most frequently used to implement emergency escape mechanisms from inside of deeply nested loops or recursive procedures. Continuations can be used to implement any of the *catch, throw, return,* or *goto* constructs of continuation-less LISPs. They have been used to implement not only coroutines [HaynFW84], but complete multi-tasking systems completely within LISP [HaynF84] [Wand80].

The control environment in a LISP implementation consists of a number of frames, one for each active function invocation. Each frame contains a return address and any variable bindings which are local to the function to which the frame belongs. Each frame can therefore construct a path from itself to the *root frame* which is the frame corresponding to a procedure invoked at LISP top-level.

This structure is identical to the structure of the lambda-binding environment, which is no coincidence since the lambda-binding environment structure was derived from an implicit understanding of the control structure of a LISP application. Continuations can therefore be implemented using techniques very similar to those used to implement closures. Early implementations captured continuations exclusively. When these implementations used the continuations as closures, they simply ignored the control information present in the continuation. Recent implementations separate the binding environment frames from control frames as an optimization. Closures do not need control information or local bindings and so can be represented somewhat more compactly than continuations.

All of the techniques of the previous section therefore apply to continuations. Continuations can be implemented as trees in garbage-collected memory, or as spaghetti or phantom

stacks.

One last subtle point to keep in mind when implementing continuations is that local bindings are shared among all continuations sharing a control frame. Not only is this consistent with the treatment of shared bindings in closures, this treatment is demanded of continuations if no distinction is to be made in the semantics of LISP between local and non-local variables.

# Chapter 9

# Conclusions

The conclusion of the author is that the following techniques are suitable for use in LISP in the context of small, real memory workstations using stock CPUs.

- The memory regained from cdr-encoding is worth the small processing overhead the algorithm entails. A compact *CONS* operation in a cdr-encoding environment is worth the effort as well, since if it is used only where performance improvements are guaranteed, it is does improve performance.

- One of the hybrid garbage collector must be used if cdr-encoding is to be carried out. The only garbage collector that will work effectively with a cdr-encoding implementation is a linearizing, compacting garbage collector. The collector best suited to cdr-encoding is the two-space compactor. This collector not only linearizes all nodes in memory every time it is run, but has the smallest processing cost of all of the collectors examined. The two-space collector however, requires that one half of memory be reserved from use. This more than negates any memory performance improvements gained from cdr-encoding.

  The only other known linearizing collectors are the LISP 2 algorithm and various hybrid algorithms. The memory overhead of the LISP 2 algorithm, again, eliminates all of the memory performance improvement of cdr-encoding. The hybrid compacting algorithms are therefore the only choice left to implement cdr-encoding. Which algorithm to use is not clear, since no performance comparison of the algorithms is available.

- Both of the hybrid algorithms require a non-copying tracing algorithm for a mark and sweep phase. The algorithm of choice for the marking phase is clearly pointer reversal, since this uses less stack space than a recursive algorithm. Even with a bit-stack though, pointer reversal still requires that storage be reserved for a stack which could potentially grow very large. When there is sufficient unused storage in small nodes, it would be best if these nodes could reserve *log(num ptrs in node)* bits so that for small nodes, the bit stack would be unnecessary. It is always possible to reserve this many bits in large nodes, since the memory cost of doing so is proportionally much

smaller for large nodes. When there is not enough space in small nodes to encode this quantity, a small bit stack must be reserved and Knuth's [Knut73] recovery procedure applied when this stack overflows.

- Encoding type information in node headers is the best memory organization model when there is sufficient waste space in nodes to contain this information.

  The fact that both hybrid algorithms use a mark and sweep pass eliminates typed pointers from consideration as a memory organization model. When node headers will occupy an unacceptable amount of memory, BIBOP must be used. BIBOP is more complex and less portable than node headers, though, and should be avoided if possible.

- To minimize the memory and processing costs of lambda-binding, the MACLISP stack appears to be the fastest method around. This method suffers from a lack of generality though, since it cannot solve the upward funarg problem. A more general, though slightly less memory and processor efficient solution is Stallman's phantom stacks. This algorithm performs slightly better than Danvy's optimized spaghetti stacks since it fragments memory less than the spaghetti stacks do.

It is not clear at this time whether or not generation scavenging or reference counts will improve processing performance without hardware assistance. The estimates published so far do not bode well for these technique when applied to LISP.

## 9.1 Future Research

Empirical tests are necessary to determine which of the hybrid algorithms performs best and to determine whether or not generation scavenging or reference counting results in a performance improvement.

This study has not addressed the question of continuous or incremental garbage collectors. It seems unlikely that these systems would improve the performance of a LISP system since they require additional processing on the part of both the LISP system and the garbage collector to maintain synchronization. These algorithms may improve perceived performance though, since they eliminate annoying pauses in execution while garbage collection takes place. This study has not addressed parallel garbage collection mechanisms either, since these mechanisms require hardware assistance (ie: another CPU) not available on most simple workstations.

Additional hybrid algorithms appear to be worth investigating as well. It may be that a new hybrid can be developed with better compaction characteristics than either of the two existing ones.

# Appendix A

# Tracing List Structure Using a Stack

```
void mark (curnode)                    /* mark tree rooted at curnode */
NODE *curnode;                         /* pointer to current node */
{
     int ptrnum;                       /* pointer number in curnode */
     extern void turn_on_mark ();      /* turn on mark bit in a node */
     extern int already_marked ();     /* is the node already marked? */
     extern int num_ptrs ();           /* number of pointers in node */
     extern NODE *get_ptr ();          /* returns nth pointer in node */


/* : check if the node is already marked */
     if (already_marked (curnode) == YES)
          return;
     turn_on_mark_bit (curnode);


/* : chase each pointer in the node */
     for (ptrnum = 1; ptrnum <= num_ptrs (curnode); ptrnum++)
          mark (get_ptr (curnode, ptrnum));
     return;
}
```

# Appendix B

# Tracing List Structure Using Pointer Reversal

```
void mark (start)                       /* mark tree rooted at curnode */
NODE *start;                            /* node to start marking from */
{
     extern void turn_on_mark ();       /* turn on mark bit in a node */
     extern int already_marked ();      /* is the node already marked? */
     extern int num_ptrs ();            /* number of pointers in node */
     extern NODE *get_ptr ();           /* returns nth pointer in node */
     extern void set_ptr ();            /* replace nth pointer in node */
     extern void push_pos ();           /* save pos info using min bits */
     extern void pop_pos ();            /* restore node position */

     int ptrnum;                        /* pointer number in curnode */
     NODE *curnode;                     /* node currently being processed */
     NODE *prevnode;                    /* previous value of curnode */
     NODE *next;                        /* next node to become curnode */


/* : mark the subtree rooted at start */
     curnode = start;
     prevnode = NIL;
     ptr_num = 0;
     while (TRUE) {


/* : descend as far in the tree as possible */
          while (TRUE) {
               if ((already_marked (curnode) == YES) || (curnode == NIL))
                    break;              /* stop descending */


/* :* mark a node and its descendants */
               turn_on_mark (curnode);
               ptr_num++;
```

```
            if (ptr_num <= num_ptrs (curnode)) {
                 if (get_ptr (curnode, ptr_num) != NIL) {
                      next = get_ptr (curnode, ptr_num);
                      set_ptr (curnode, ptr_num, prev);  /* rev ptrs */
                      push_pos (curnode, ptr_num);       /* save pos */
                      prev = curnode;
                      curnode = next;
                      ptr_num = 0;
                      }
                 }

            /* no more descendants - stop descending */
            else break;
            }


/* : back up to a point where further descent is possible */
          while (TRUE) {

               /* no more backing up is possible, the subtree is traced */
               if (prev == NIL)
                    return;

/* :* back up a step */
               nth_ptr = pop_pos (prev);
               next = get_ptr (prev, nth_ptr);    /* get reversed ptr */
               set_ptr (prev, nth_ptr, curnode);  /* restore list */
               curnode = prev;
               prev = next;

/* :* stop backing up when a node can be traced further */
               if (nth_ptr < num_ptrs (curnode))
                    break;
               }
          }
}

/* push a number on the stack using the minimum number of bits */
void push_pos (curnode, position)
NODE *curnode;                          /* node that position applies to */
int position;                           /* pointer position in node */
{
     extern int num_ptrs ();            /* number of pointers in node */
     extern void push_bits ();          /* push low N bits of arg */
     extern int log2N ();               /* minimum number of bits to use */

     push_bits (position - 1, log2N (num_ptrs (curnode)));
}

/* pop a number from the stack using the minimum number of bits */
int pop_pos (curnode)
```

```
NODE *curnode;                          /* node that position applies to */
{
     extern int num_ptrs ();            /* number of pointers in node */
     extern void push_bits ();          /* push low N bits of arg */
     extern int log2N ();               /* minimum number of bits to use */
     int N;                             /* number of bits to pop */

     N = log2N (num_ptrs (curnode));
     if (N == 0)
          return (1);                   /* 1 pointer needs no bits */
     else return (pop_bits (log2N (num_ptrs (curnode))) + 1);
}
```

# Appendix C

# Tracing By Moving

```
NODE *trace_by_moving (start)          /* move tree of nodes */
NODE *start;                           /* the node to move first */
{
     extern void copy_node ();         /* copy a node */
     extern int fwding_ptr ();         /* is the node already moved? */
     extern int num_ptrs ();           /* number of pointers in node */
     extern NODE *get_ptr ();          /* returns nth pointer in node */
     extern void set_ptr ();           /* replace nth pointer in node */
     extern int node_size ();          /* return size of node in bytes */
     extern char *new_space ();        /* get big, empty memory region */

     int ptrnum;                       /* pointer number in curnode */
     NODE *curptr;                     /* current pointer in curnode */
     NODE *curnode;                    /* node currently being processed */
     char *next_free;                  /* next free pos in new-space */
     char *rescan;                     /* nodes to pointer-adjust */


/* : acquire new space */
     next_free = new_space ();
     rescan = next_free;


/* : move the root node to new space */
     copy_node (start, next_free);
     set_ptr (start, 1, next_free);    /* leave fwding ptr behind */
     start = next_free;                /* remember where we moved start */
     next_free += node_size (start);


/* : adjust all pointers in new space */
     while (rescan != next_free) {
          curnode = (NODE *) rescan;
          for (ptrnum = 1; ptrnum < num_ptrs (curnode); ptrnum++) {
               curptr = get_ptr (curnode, ptr_num);
```

```
/* :* pointers to forwarding pointers are simply replaced */
               if (fwd_ptr (get_ptr (curptr, 1)) == YES)
                    set_ptr (curnode, ptr_num, get_ptr (curptr, 1));


/* :* unmoved nodes are moved & the new address is the new pointer */
               else {
                    copy_node (curptr, next_free);      /* copy node */
                    set_ptr (curptr, 1, next_free);     /* set fwding ptr */
                    set_ptr (curnode, ptr_num, next_free);  /* adjust ptr */
                    next_free += node_size ((NODE *) curptr);
                    }
               }                          /* end of for all pointers */


/* : make the rescan pointer point at the next node in new-space */
          rescan += node_size ((NODE *) rescan);
          }                               /* end of adjust all pointers */
     return (start);                      /* return new location of start */
}
```

# Appendix D

# Optimized Tracing By Moving

```
NODE *trace_by_moving (start)          /* move all nodes in tree */
NODE *start;                           /* the node to move first */
{
     extern void copy_node ();         /* copy a node */
     extern int fwding_ptr ();         /* is the node already moved? */
     extern int num_ptrs ();           /* number of pointers in node */
     extern NODE *get_ptr ();          /* returns nth pointer in node */
     extern void set_ptr ();           /* replace nth pointer in node */
     extern int node_size ();          /* return size of a node in bytes */
     extern char *new_space ();        /* get big, empty memory region */

     int ptrnum;                       /* pointer number in curnode */
     NODE *curptr;                     /* current pointer in curnode */
     NODE *curnode;                    /* node currently being processed */
     NODE *new_node;                   /* where to copy curnode to */
     char *next_free;                  /* next free pos in new-space */
     NODE *rescan;                     /* nodes to pointer-adjust */
     NODE *cs_result;                  /* result of copying list */


/* : acquire new space */
     next_free = new_space ();
     rescan = NIL;


/* : copy the start node to new space */
     new_node = (NODE *) next_free;
     next_free = next_free + node_size (start);
     copy_node (start, new_node);
     set_ptr (start, 1, new_node);     /* leave fwding pointer behind */
     start = new_node;                 /* remember where start was moved */


/* :* check if the start node needs to be added to the rescan list */
     if (num_ptrs (start) > 1) {
          set_ptr (start, 2, rescan);
```

```
          rescan = start;
          }

/* :* check if the start node is too small to add to the rescan list */
     else if (num_ptrs (start) == 1)
          copy_singles (new_node, &next_free, &rescan);


/* : rescan everything on the rescan list */
     while (rescan != NIL) {
          curnode = get_ptr (rescan, 1);
          rescan = get_ptr (rescan, 2);
          for (ptrnum = 1; ptrnum < num_ptrs (curnode); ptrnum++) {
               curptr = get_ptr (curnode, ptr_num);


/* :* simply replace pointers to forwarding pointers */
               if (fwd_ptr (get_ptr (curptr, 1)) == YES)
                    set_ptr (curnode, ptr_num, get_ptr (curptr, 1));


/* :* copy nodes which have not yet been forwarded */
               else {
                    new_node = (NODE *) next_free;
                    next_free += node_size (curnode);
                    copy_node (curptr, new_node);      /* copy node */
                    set_ptr (curptr, 1, new_node);     /* fwding ptr */
                    set_ptr (curnode, ptr_num, new_node);   /* adjust ptr */


/* :** check if the copied node must be added to the rescan list */
                    if (num_ptrs (new_node) > 1) {
                         set_ptr (curptr, 2, rescan);
                         rescan = curptr;
                         }


/* :** check if the copied node is too small to add to the rescan list */
                    else if (num_ptrs (new_node) == 1)
                         copy_singles (new_node, &next_free, &rescan);
                    }                    /* end of copy all other nodes */
               }                         /* end of for all pointers */
          }                              /* end of rescan the rescan list */
     return (start);                     /* return new copy of start */
}

/* Curnode contains a single pointer field and so cannot be added to the */
/* rescan list.  Chase that pointer and copy the node it points at to */
/* new space.  Continue chasing all pointers as long as their targets are */
/* nodes containing a single pointer */
```

```
void copy_singles (curnode, next_free, rescan)
NODE *curnode;                          /* single pointer node to chase */
char **next_free;                       /* start of free new-space memory */
NODE **rescan;                          /* start of rescan list */
{
     NODE *curptr;                      /* the pointer in the node */

/* : replace forwarding pointers */
     do {
          curptr = get_ptr (curnode, 1);
          if (fwd_ptr (get_ptr (curptr, 1))) == YES)
               set_ptr (curnode, 1, get_ptr (curptr, 1));


/* : copy nodes which have not yet been forwarded */
          else {
               new_node = (NODE *) next_free;
               new_node = (NODE *) *next_free;
               *next_free += node_size (curnode);
               copy_node (curptr, new_node);      /* copy node */
               set_ptr (curptr, 1, new_node);     /* fwding ptr */
               set_ptr (curnode, ptr_num, new_node);   /* adjust ptr */


/* : check if the copied node must be added to the rescan list */
               if (num_ptrs (new_node) > 1) {
                    set_ptr (curptr, 2, rescan);
                    rescan = curptr;
                    }
               }
          }


/* : check if the copied node is too small to add to the rescan list */
     while (num_ptrs (new_node) == 1);
     return;
}
```

# Appendix E

# Jonkers' Sliding Compaction Algorithm

```
void compact (rootnode, start_of_memory, memory_size)
NODE *rootnode;                          /* root of tree of used nodes */
char *start_of_memory;                   /* the lowest address in memory */
long memory_size;                        /* the number of bytes in memory */
{
     int ptrnum;                         /* pointer number in curnode */
     NODE *curptr;                       /* current pointer in curnode */
     NODE *curnode;                      /* the node being processed */
     char *newpos;                       /* new position for this node */

     extern void ptr_reversal_mark ();   /* ptr-reverse-mark tree of nodes */
     extern int is_marked ();            /* is the node marked? */
     extern int num_ptrs ();             /* number of pointers in node */
     extern NODE *get_ptr ();            /* returns nth pointer in node */
     extern void set_ptr ();             /* replace nth pointer in node */
     extern int node_size ();            /* size of a node in bytes */
     extern void chain ();               /* add a node to reference chain */
     extern void unchain ();             /* adjust ptrs in reference chain */


/* : pass 1 - pointer reversal marking */
     ptr_reversal_mark (rootnode);


/* : pass 2 - visit every used node in memory */
     newpos = start_of_memory;
     for (curnode = (NODE *) start_of_memory;
          curnode < (NODE *) (start_of_memory + memory_size);
          curnode = (NODE *) (((char *) curnode) + node_size (curnode))) {
          if (is_marked (curnode)) {


/* :* do pointer adjustments of all forward pointers to this node */
```

```
                unchain (curnode, newpos);


/* :* prepare each pointer in the node for future pointer adjustment */
                for (ptrnum = 0; ptrnum < num_ptrs (curnode); ptrnum++) {

                     /* curptr points to the node containing it? */
                     if (get_ptr (curnode, ptrnum) == curnode)
                          set_ptr (curnode, ptrnum, new_node);

                     /* curptr is either a forward or a backwards pointer */
                     else chain (curnode, ptrnum);
                     }


/* :* adjust predicted location of next used node */
                newpos += node_size (curnode);
                }
          }


/* : pass 3 - visit every used node in memory */
     newpos = start_of_memory;
     for (curnode = (NODE *) start_of_memory;
          curnode < (NODE *) (start_of_memory + memory_size);
          curnode = (NODE *) (((char *) curnode) + node_size (curnode))) {
          if (is_marked (curnode)) {


/* :* copy the node to its new home */
                copy_node (curnode, newpos);


/* :* do pointer adjustment on all backwards pointers to this node */
                unchain (newpos, newpos);


/* :* adjust predicted location of next used node */
                newpos += node_size (curnode);
                }
          }
     return;
}

/* chain a pointer to a node */
void chain (curnode, ptrnum);
NODE *curnode;                          /* node containing the pointer */
int ptrnum;                             /* index of ptr to chain */
{
     NODE **nodeptraddr;                /* address of pointer in node */
     long oldcontents;                  /* contents of ptr-sized field */
```

```
     int need_end_of_chain;             /* need to set end of chain bit */

     extern char **get_ptr_addr ();     /* return addr of nth ptr in node */
     extern void set_ptr ();            /* replace nth pointer in node */
     extern long get_ptr_sized_field ();/* get value of ptr-sized field */
     extern void set_ptr_sized_field ();/* sets ptr-sized field */
     extern int ptr_sized_field_was_set ();  /* was ptr-sized field set? */
     extern void set_end_of_chain ();   /* indicate target of ptr is eoc */


/* : chain ptr-sized field to the location containing current ptr */
     nodeptraddr = get_ptr_addr (curnode, ptrnum);
     oldcontents = get_ptr_sized_field (*nodeptraddr);
     if (ptr_sized_field_was_set (*nodeptraddr) == NO) {
          set_ptr_sized_field (set_end_of_chain (nodeptraddr));
     else set_ptr_sized_field (nodeptraddr);


/* : replace the current pointer with old contents of ptr target */
     *nodeptraddr = oldcontents;
     return;
}

/* adjust pointers in a chain of references to a location */
void unchain (curnode, newpos)
NODE *curnode;                          /* node which is start of chain */
NODE *newpos;                           /* new position for node */
{
     NODE **curptraddr;                 /* pointer to pointer to adjust */
     NODE **nextptraddr;                /* ptr to next ptr to adjust */

     extern long get_ptr_sized_field ();/* get value of ptr-sized field*/
     extern void set_ptr_sized_field ();/* sets ptr-sized field */
     extern void reset ();              /* ptr-sized field is not set */
     extern int ptr_sized_field_was_set ();  /* was ptr-sized field set? */
     extern void end_of_chain ();       /* indicate target of ptr is eoc */


/* : check if there's any unchaining to do at all */
     if (ptr_sized_field_was_set (curnode) == YES) {


/* : keep adjusting pointers until the end of chain looms */
          for (curptraddr = (NODE **) get_ptr_sized_field (curnode);
               end_of_chain (curptraddr) == NO;
               curptraddr = (NODE **) nextptraddr) {

               nextptraddr = (NODE **) *curptraddr;    /* save chain ptr */
               *curptraddr = newpos;                   /* adjust ptr */
               }
```

```
/* : adjust the end of the list */
          *curptraddr = newpos;                          /* adjust last ptr */
          set_ptr_sized_field (curnode, *curptraddr);    /* restore field */
          reset (curnode);                               /* reset indicator */
          }
     return;
}
```

# Appendix F

# Lang and Dupont's Hybrid Incrementally Compacting Algorithm

```
/* NOTES

The procedure update_stats is not included here.  The question
"What is a good heuristic for determining which region of
memory to designate as old-space?" is still open.
*/


/* information describing memory (containing a single compaction region) */
struct memory {
     char *low;                          /* lowest address in memory */
     char *high;                         /* highest address in memory */
     char *lold;                         /* lowest address old space */
     char *hold;                         /* highest address in old space */
     char *lnew;                         /* lowest address in new space */
     char *hnew;                         /* highest address in new space */
     };

/* the structure of an item on the free list */
struct free_item {
     long size;                          /* number of bytes item */
     struct free_item *next;             /* next free item in list */
     };


/* the hybrid garbage collector */
NODE *collect (mem, root)                /* returns the free list */
NODE *root;                              /* root node of used node tree */
struct memory *mem;                      /* description of memory */
{
     extern NODE *markall ();            /* mark all of memory */
     extern struct fstats *sweep ();     /* sweep memory, return stats */
     extern void add_free ();            /* add a region to free list */
```

```
     extern struct free_item *free_list;     /* the free memory blocks */
     struct fstats stats;               /* statistics about the free list */
     NODE *next_free;                   /* next free loc in new-space */


/* : mark all of memory */
     next_free = markall (root, mem);


/* : sweep all of mark-and-sweep memory */
     sweep (mem, &free_list, &stats);


/* : add the unused portion of new-space to the free list */
     add_free (next_free, mem -> hnew - next_free + 1, &free_list);


/* : move the compaction region to its new home */
     if (stats.resize == NO) {
          mem -> lnew = mem -> lold;    /* just use old-space */
          mem -> hnew = mem -> hold;
          }
     else {
          mem -> lnew = stats.lold;     /* use what stats says to */
          mem -> hnew = stats.hold;
          }
     mem -> lold = stats.lold;
     mem -> hold = stats.hold;

     return (free_list);
}


/* mark all of the used nodes in memory */
NODE *markall (mem, root)
struct memory mem;                      /* what memory looks like */
NODE *root;                             /* the root of all in-use nodes */
{
     extern int num_ptrs ();            /* number of pointers in node */
     extern NODE *get_ptr ();           /* returns nth pointer in node */
     extern void set_ptr ();            /* replace nth pointer in node */
     extern int node_size ();           /* size of a node in bytes */
     extern void mark ();               /* pointer-reverse mark a subtree */

     int ptrnum;                        /* pointer number in curnode */
     NODE *curptr;                      /* current pointer in curnode */
     NODE *curnode;                     /* node currently being processed */
     char *next_free;                   /* next free pos in new-space */
     char *rescan;                      /* list of nodes to ptr-adjust */
```

```
/* : INIT */
     next_free = mem -> lnew;
     rescan = next_free;

/* : mark the root node */
     mark (root, mem, &next_free);

/* : adjust all pointers in all nodes new space */
     while (rescan != next_free) {
          curnode = (NODE *) rescan;
          for (ptrnum = 1; ptrnum < num_ptrs (curnode); ptrnum++) {
               curptr = get_ptr (curnode, ptr_num);
               mark (curptr, mem, &next_free);
               if (in_old_space (curptr, mem) == YES)
                    set_ptr (curnode, ptr_num, get_ptr (curptr, 1));
               }                          /* end of for all pointers */
          }                               /* end of adjust all nodes */
     return (next_free);
}


/* pointer reversal tracing */
void mark (start, mem, next_free)        /* mark reachable nodes */
NODE *start;                             /* node to start marking from */
struct memory *mem;                      /* what memory looks like */
char **next_free;                        /* next free pos in new-space */
{
     extern void turn_on_mark ();        /* turn on mark bit in a node */
     extern int already_marked ();       /* is the node already marked? */
     extern int num_ptrs ();             /* number of pointers in node */
     extern NODE *get_ptr ();            /* returns nth pointer in node */
     extern void set_ptr ();             /* replace nth pointer in node */
     extern void push_pos ();            /* save pos info using min bits */
     extern void pop_pos ();             /* restore node position */
     extern void copy_node ();           /* copy a node */
     extern int fwding_ptr ();           /* is the node already moved? */

     int ptrnum;                         /* pointer number in curnode */
     NODE *curnode;                      /* node being processed */
     NODE *prevnode;                     /* previous value of curnode */
     NODE *next;                         /* next value of curnode */


/* : mark the subtree rooted at start */
     curnode = start;
     prevnode = NIL;
     ptr_num = 0;
     while (TRUE) {
```

```
/* : descend as far in the tree as possible */
          while (TRUE) {


/* : copy nodes in old-space to new-space and stop descending */
               if (in_old_space (curnode, mem) == YES) {
                    if (fwding_ptr (curnode, mem) == NO) {
                         copy_node (curnode, *next_free);
                         set_ptr (curnode, 1, *next_free);
                         }
                    break;              /* stop descending */
                    }


/* : nodes already marked mean stop descending */
               if ((already_marked (curnode) == YES) || (curnode == NIL))
                    break;              /* stop descending */


/* :* mark a node and its descendants */
               turn_on_mark (curnode);
               ptr_num++;
               if (ptr_num <= num_ptrs (curnode)) {
                    if (get_ptr (curnode, ptr_num) != NIL) {
                         next = get_ptr (curnode, ptr_num);
                         set_ptr (curnode, ptr_num, prev);  /* rev ptrs */
                         push_pos (curnode, ptr_num);       /* save pos */
                         prev = curnode;
                         curnode = next;
                         ptr_num = 0;
                         }
                    }

               /* no more descendants - stop descending */
               else break;
               }


/* : back up to a point where further descent is possible */
          while (TRUE) {

               /* no more backing up is possible, the subtree is traced */
               if (prev == NIL)
                    return;

/* :* back up a step */
               nth_ptr = pop_pos (prev);
               next = get_ptr (prev, nth_ptr);    /* get reversed ptr */
               set_ptr (prev, nth_ptr, curnode);  /* restore list */
               curnode = prev;
```

```
                prev = next;

/* :* stop backing up when a node can be traced further */
                if (nth_ptr < num_ptrs (curnode))
                     break;
                }
           }
}


/* determine whether or not a node is in old space */
int in_old_space (curnode, mem)
NODE *curnode;                          /* the node in question */
struct memory *mem;                     /* what memory looks like */
{
     return ((curnode >= mem -> lold) && (curnode <= mem -> hold));
}


/* sweep all of ms-space */
void sweep (mem, free_list, stats);
struct memory *mem;                     /* new description of memory */
struct free_item **free_list;           /* next free pos in new-space */
struct fstats *stats;                   /* fragmentation statistics */
{
     extern void sweep_region ();       /* sweep a region of memory */
     extern void update_stats ();       /* update fragmentation stats */


/* : INIT */
     *free_list = NIL;


/* : sweep all of ms-space, adding unused stuff to the free list */
     if (mem -> lold == NIL)            /* is all of memory ms-space? */
          sweep_region (mem -> lold, mem -> hold, free_list, stats);
     else {
          sweep_region (mem -> low,
               min (mem -> lold, mem -> hold) - 1, free_list, stats);
          sweep_region (min (mem -> hold, mem -> hnew) + 1,
               max (mem -> lold, mem -> lnew) - 1, free_list, stats);
          sweep_region (max (mem -> hold, mem -> hnew) + 1,
               mem -> high, free_list, stats);
          }


/* : tell the statistics subsystem that the sweep of ms-space is complete */
     update_stats (stats, mem, NIL);
     return (stats);
}
```

```
/* sweep a region of memory */
void sweep_region (low, high, free_list, stats)
char *low;                              /* lowest address in region */
char *high;                             /* highest address in region */
struct free_item **free_list;           /* current free list */
struct fstat *stats;                    /* fragmentation statistics */
{
     extern int already_marked ();      /* is the node already marked? */
     extern int node_size ();           /* size of a node in bytes */
     extern void add_free ();           /* add something to free list */
     extern void update_stats ();       /* update fragmentation stats */

     NODE *curnode;                     /* the node being examined */
     NODE *freenode;                    /* a free node */


/* : visit every node in the region */
     curnode = (NODE *) low;
     while (curnode < (NODE *) high) {


/* : skip in-use nodes
          while ((already_marked (curnode) == YES) &&
               (curnode < (NODE *) high))
               ((char *) curnode) += node_size (curnode);


/* : add ununsed nodes to the free list */
          freenode = curnode;
          while ((already_marked (curnode) == NO) &&
               (curnode < (NODE *) high))
               ((char *) curnode) += node_size (curnode);
          if (freenode < (NODE *) high) {
               add_free (curnode, curnode - freenode, fl);
               update_stats (stats, mem, freenode, curnode - freenode);
               }
          }
     return;
}

/* add a region of memory to the free list */
void add_free (item, size, free_list);
struct free_item *item;                 /* item to add to free list */
long size;                              /* size of item */
struct free_item **free_list;           /* the current free list */
{


/* : check if the item is big enough */
     if (size >= sizeof (struct free_item)) {
```

```
            item -> size = size;
            item -> next = *free_list;
            *free_list = item;
            }
        return;
}
```

# Appendix G

# Ginter's Hybrid Incrementally Compacting Algorithm

```
/* NOTES

The procedure update_stats is not included here.  The question
"What is a good heuristic for determining which region of
memory to designate as old-space?" is still open.
*/


/* information describing memory */
struct memory {
     char *low;                          /* lowest address in memory */
     char *high;                         /* highest address in memory */
     char *lold;                         /* lowest address in old-space */
     char *hold;                         /* highest address in old-space */
     };

/* the structure of an item on the free list */
struct free_item {
     struct free_item *next;             /* next free item in list */
     long size;                          /* size of this free item */
     };

/* the structure of the free list */
struct free_list {
     struct free_item *first;            /* first entry in the list */
     struct free_item *last;             /* last entry in the list */
     };

/* the structure of a sequential position in the free list */
struct free_pos {
     struct free_item *current;          /* current free list entry */
     char *end;                          /* the end of the current entry */
     char *curpos;                       /* current position in this entry */
```

```
    char *virtpos;                      /* virtual position in new space */
    };


/* the hybrid garbage collector */
NODE *collect (mem, root)               /* returns the free list */
NODE *root;                             /* root of the used node tree */
struct memory *mem;                     /* description of memory */
{
    extern struct free_item markall ();     /* mark all of memory */
    extern void sweep ();               /* sweep all of memory */
    extern void copy_free_list ();      /* copy free list to old space */

    extern struct free_list ();         /* list of free blocks of memory */
    struct fstats stats;                /* statistics about the free list */
    struct free_item *first_fwd;        /* first entry to chain forwards */


/* : the "normal" mark and sweep pass */
    mem -> old = NIL;                   /* first pass - no old space */
    mem -> hold = NIL;
    markall (root, mem, NIL);
    sweep (mem, &fl, &stats);


/* : the second marking pass - to old space to the free list */
    mem -> old = stats.hold;
    mem -> hold = stats.hold;
    first_fwd = markall (root, mem, &fl);


/* : copy the free list back int  old space & rebuild the free list */
    copy_free_list (fl, mem, first_fwd);
    return (free_list);
}


/* return the size of a node */
int node_size (curnode)
NODE *curnode,                          /* the node whose size we want */
{
    extern int node_type ();            /* returns the type of a node */
    extern int typetable ();            /* size of that node type */

    return (typetable[node_type (curnode)]);
}

/* return the number of pointers in a node */
int num_ptrs (curnode)
NODE *curnode,                          /* the node we're interested in */
{
```

```
     extern int node_type ();           /* returns the type of a node */
     extern int type2num ();            /* number of ptrs in node type */

     return (type2num (node_type (curnode)));
}

/* return the Nth pointer in a node */
NODE *get_ptr (curnode, n)
NODE *curnode;                          /* the node we're interested in */
int n;                                  /* the ptr we're interested in */
{
     extern int node_type ();           /* return type of a node */
     extern int ptroffset ();           /* offset of ptr in that type */

     return (*((NODE **) (((char *) curnode) +
          (ptroffset (node_type (curnode), n)))));
}

/* set the Nth pointer in a node */
void set_ptr (curnode, n, newval)
NODE *curnode;                          /* the node we're interested in */
int n;                                  /* the ptr we're interested in */
NODE *newval;                           /* new value for that pointer */
{
     extern int node_type ();           /* determine the type of a node */
     extern int ptroffset ();           /* offset of ptr in that type */

     int offset;                        /* the offset of the pointer */

     offset = ptroffset (node_type (curnode), n);
     *((NODE **) (((char *) curnode) + offset)) = newval;
     return;
}


/* mark all of the used nodes in memory */
struct free_item *markall (mem, root, fl)
struct memory mem;                      /* what memory looks like */
NODE *root;                             /* the root of all in-use nodes */
struct free_list *fl;                   /* the free list */
{
     extern int node_type ();           /* determine the type of a node */
     extern int type2num ();            /* number of ptrs in that type */
     extern int ptroffset ();           /* offset of ptr in that type */
     extern int type2size ();           /* size of a type of node */
     extern void read_bytes ();         /* read from free-list */
     extern struct free_item *write_bytes ();     /* write to free-list */
     extern void move_to_pos ();        /* move to a virtual position */
     extern void init_pos ();           /* init free_pos data structure */
     extern struct free_item *mark ();  /* ptr-reverse mark a subtree */
```

```
     int ptrnum;                        /* pointer number in curnode */
     NODE *curptr;                      /* current pointer in curnode */
     char *curnode;                     /* virtual pos of current node */
     struct fp next_free;               /* next free pos in new-space */
     struct fp rescan_read;             /* read here for ptr-adjusting */
     struct fp rescan_write;            /* write here for ptr-adjusting */
     int curtype;                       /* type of curnode */
     int offset;                        /* current offset in curnode */
     struct free_item *first_fwd;       /* first entry to chain fwds */
     struct free_item *result;          /* result of mark */


/* : INIT */
     if (mem -> lold != NIL) {
          init_pos (&next_free, fl, mem);
          init_pos (&rescan_read, fl, mem);
          init_pos (&rescan_write, fl, mem);
          }
     first_fwd = NIL;


/* : mark the root node */
     result = mark (root, mem, &next_free);
     if (result != NIL)
          first_fwd = result;


/* : adjust all pointers in all nodes in new space */
     for (curnode = mem -> lold; curnode < next_free.virtpos;
          curnode += type2size (curtype)) {
          move_pos (&rescan_read, curnode);
          curtype = node_type (rescan_read.curpos);
          for (ptrnum = 1; ptrnum < type2num (curtype); ptrnum++) {


/* : * get the current pointer */
               offset = ptroffset (curtype, ptrnum);
               move_pos (&rescan_read, curnode + offset);
               read_bytes (&rescan_read, &curptr, sizeof (curptr));


/* : mark it and adjust it *:/
               result = mark (curptr, mem, &next_free);
               if (result != NIL)
                    first_fwd = result;
               if (in_old_space (curptr, mem) == YES) {
                    curptr = get_ptr (curptr, 1);
                    move_pos (&rescan_write, curnode + offset);
                    write_bytes (&rescan_write, &curptr, sizeof (curptr));
                    }
               }                        /* end of for all pointers */
```

```
        }                               /* end of adjust all nodes */
    return;
}


/* pointer reversal tracing */
struct free_item *mark (start, mem, next_free)
NODE *start;                            /* node to start marking from */
struct memory *mem;                     /* what memory looks like */
struct free_pos *next_free;             /* next free pos in new-space */
{
    extern void turn_on_mark ();        /* turn on mark bit in a node */
    extern int already_marked ();       /* is the node already marked? */
    extern int num_ptrs ();             /* number of pointers in node */
    extern NODE *get_ptr ();            /* returns nth pointer in node */
    extern void set_ptr ();             /* replace nth pointer in node */
    extern void push_pos ();            /* save pos using minimum bits */
    extern void pop_pos ();             /* restore node position */
    extern int fwding_ptr ();           /* is the node already moved? */
    extern struct free_item *write_bytes ();     /* write to free space */
    extern void move_pos ();            /* move to a virtual position */

    int ptrnum;                         /* pointer number in curnode */
    NODE *curnode;                      /* node being processed */
    NODE *prevnode;                     /* previous value of curnode */
    NODE *next;                         /* node to process next */
    struct free_item *first_back;       /* first node to chain bkwds */
    struct free_item *result;           /* result of write_bytes op */


/* : INIT */
    first_back = NIL;


/* : mark the subtree rooted at start */
    curnode = start;
    prevnode = NIL;
    ptr_num = 0;
    while (TRUE) {


/* : descend as far in the tree as possible */
        while (TRUE) {


/* : copy nodes in old-space to new-space and stop descending */
            if (in_old_space (curnode, mem) == YES) {
                if (fwding_ptr (curnode, mem) == NO) {
                    result = write_bytes (next_free, curnode,
                        node_size (curnode));
                    if (result != NIL)
```

```
                    first_back = result;
                set_ptr (curnode, 1, next_free -> virtpos);
                }
            break;              /* stop descending */
            }


/* : nodes already marked mean stop descending */
        if ((already_marked (curnode) == YES) || (curnode == NIL))
            break;              /* stop descending */


/* :* mark a node and its descendants */
        turn_on_mark (curnode);
        ptr_num++;
        if (ptr_num <= num_ptrs (curnode)) {
            if (get_ptr (curnode, ptr_num) != NIL) {
                next = get_ptr (curnode, ptr_num);
                set_ptr (curnode, ptr_num, prev);  /* rev ptrs */
                push_pos (curnode, ptr_num);       /* save pos */
                prev = curnode;
                curnode = next;
                ptr_num = 0;
                }
            }

        /* no more descendants - stop descending */
        else break;
        }


/* : back up to a point where further descent is possible */
    while (TRUE) {

        /* no more backing up is possible, the subtree is traced */
        if (prev == NIL)
            return (first_back);

/* :* back up a step */
        nth_ptr = pop_pos (prev);
        next = get_ptr (prev, nth_ptr);    /* get reversed ptr */
        set_ptr (prev, nth_ptr, curnode);  /* restore list */
        curnode = prev;
        prev = next;

/* :* stop backing up when a node can be traced further */
        if (nth_ptr < num_ptrs (curnode))
            break;
        }
    }
}
```

```
/* determine whether or not a node is in old space */
int in_old_space (curnode, mem)
NODE *curnode;                          /* the node in question */
struct memory *mem;                     /* what memory looks like */
{
     return ((curnode >= mem -> lold) && (curnode <= mem -> hold));
}


/* sweep all of ms-space */
void sweep (mem, fl, stats);
struct memory *mem;                     /* new description of memory */
struct free_list *fl;                   /* the free list */
struct fstats *stats;                   /* fragmentation statistics */
{
     extern int already_marked ();      /* is the node already marked? */
     extern int node_size ();           /* number of bytes in node */
     extern void add_free ();           /* add a region to free list */
     extern void update_stats ();       /* update fragmentation stats */

     NODE *curnode;                     /* the node being examined */
     NODE *freenode;                    /* a free node */


/* : visit every node in the region */
     curnode = (NODE *) (mem -> low);
     while (curnode < (NODE *) (mem -> high)) {


/* : skip in-use nodes
          while ((already_marked (curnode) == YES) &&
               (curnode < (NODE *) (mem -> high)))
               ((char *) curnode) += node_size (curnode);


/* : add ununsed nodes to the free list */
          freenode = curnode;
          while ((already_marked (curnode) == NO) &&
               (curnode < (NODE *) (mem -> high)))
               ((char *) curnode) += node_size (curnode);
          if (freenode < (NODE *) (mem -> high)) {
               add_free (curnode, curnode - freenode, fl);
               update_stats (stats, mem, freenode, curnode - freenode);
               }
          }


/* : determine where old-space should be located */
     update_stats (stats, mem, NIL, 0L);
```

```
     return;
}


/* copy the contents of the free list back into old-space */
void copy_free_list (fl, mem, first_fwd)
struct free_list *fl;                    /* the free list */
struct memory *mem;                      /* what memory looks like */
struct free_item *first_fwd;             /* first entry to chain forwards */
{
     extern void memcpy ();              /* copy memory */
     struct free_item *curitem;          /* current free item */
     struct free_item *previtem;         /* previous free item */
     struct free_item *nextitem;         /* next free item */
     struct free_item *first_back;       /* beginning of backwards chain */
     struct free_item *first_fwd;        /* beginning of forwards chain */
     char *curpos;                       /* cur virt pos in old-space */
     int cursize;                        /* size of current item */
     char *fwdpos;                       /* pos to start forward copying */


/* : search for the last free entry to copy backwards */
     previtem = NIL;
     curpos = mem -> lold;
     for (curitem = fl -> first; curitem != first_fwd; curitem = nextitem) {
          curpos += curitem -> size - (sizeof struct free_item);
          nextitem = curitem -> next;    /* reverse links for bkwd copy */
          curitem -> next = previtem;
          previtem = curitem;
          }
     first_back = previtem;
     first_fwd = curitem;
     fwdpos = curpos;


/* : copy backwards each of the entries which must be copied backwards */
     for (curitem = first_back; curitem != NIL; curitem = curitem -> next) {
          cursize = curitem -> size - sizeof (struct free_item);
          memcpy (curpos - cursize, ((char *) curitem) + cursize, cursize);
          curpos -= cursize;
          }


/* : copy forwards each of the entries which must be copied forwards */
     curpos = fwdpos;
     for (curitem = first_fwd; curitem != NIL; curitem = curitem -> next) {
          cursize = curitem -> size - sizeof (struct free_item);
          memcpy (curpos, ((char *) curitem) + cursize, cursize);
          curpos += cursize;
          }
```

```
/* : restore the free list */
     if (first_back != NIL) {
          if (first_fwd == NIL)
               fl -> last = fl -> first;
          fl -> first -> next = first_fwd;
          fl -> first = first_back;
          }
     return;
}


/* add a region of memory to the free list */
void add_free (item, size, fl);
struct free_item *item;                 /* item to add to free list */
long size;                              /* size of item */
struct free_list *fl;                   /* the current free list */
{


/* : check if the item is big enough to put on the free list */
     if (size >= sizeof (struct free_item)) {
          item -> size = size;
          if (fl -> last != NIL)
               fl -> last -> next = item;
          item -> next = NIL;
          fl -> last = item;
          if (fl -> first == NIL)
               fl -> first = item;
          }
     return;
}

/* initialize a free_pos data structure */
void init_pos (pos, fl, mem)
struct free_pos *pos;                   /* the thing to initialize */
struct free_list *fl;                   /* the free list */
struct memory *mem;                     /* what memory looks like */
{
     struct free_item *curitem;         /* current free item */


/* : find the first free element large enough to hold additional info */
     for (curitem = fl -> first;
          curitem -> size == sizeof (struct free_item);
          curitem = curitem -> next);


/* : initialize the data structure */
     pos -> current = (char *) curitem;
     pos -> end = pos -> current + curitem -> size;
```

```
     pos -> curpos = pos -> current + sizeof (struct free_item);
     pos -> virt_pos = mem -> lold;
     return;
}

/* seek forward to the desired virtual position */
void move_pos (pos, destpos)
struct free_pos *pos;                    /* current position in free list */
char *destpos;                           /* desired virtual position */
{
     int delta;                          /* how much to adjust position */
     struct free_item *curitem;          /* current unused item */


/* : is the desired location within the current unused item? */
     delta = destpos - pos -> virtpos;
     if (pos -> curpos + delta < pos -> end)
          pos -> curpos += delta;


/* : search for the unused item containing the desired location */
     else {
          delta -= (pos -> end - pos -> curpos);
          curitem = pos -> current -> next;
          while (delta > (curitem -> size - sizeof (struct free_item))) {
               delta -= curitem -> size - sizeof (struct free_item);
               curitem = curitem -> next;
               }


/* : make that item the current item */
          pos -> current = curitem;
          pos -> end = ((char *) curitem) + curitem -> size;
          pos -> curpos = ((char *) curitem) + curitem -> delta;
          pos -> virtpos = destpos;
          }
     return;
}

/* write information sequentially to unused storage - return the first */
/* item which must be processed forwards rather than backwards */
struct free_item *write_bytes (pos, from, bytes)
struct free_pos *pos;                    /* current position in free list */
char *from;                              /* what to write */
int bytes;                               /* how much to write */
{
     extern void memcpy ();              /* copy memory */

     int backwards;                      /* started with bkwds stuff */
     struct free_item *last_back;        /* the last backwards item */
     struct free_item *prev;             /* previous unused item */
```

```
     int cursize;                        /* num bytes to copy this time */


/* : INIT */
     last_back = NIL;


/* : remember whether we're processing forwards or backwards at the start */
     if (pos -> virtpos > pos -> curpos)
          backwards = YES;
     else backwards = NO;


/* : write as much as possible into the current unused item */
     cursize = pos -> end - pos -> curpos;
     if (cursize > bytes)
          cursize = bytes;
     if (cursize > 0)
          memcpy (pos -> curpos, from, cursize);
     pos -> curpos += cursize;
     pos -> virtpos += cursize;
     from += cursize;
     bytes -= cursize;


/* : keep writing to additional items until we're finished writing */
     while (bytes > 0) {
          prev = pos;
          pos -> current = pos -> current -> next;
          pos -> curpos = ((char *) pos -> current) +
               sizeof (struct free_item);
          if ((pos -> curpos <= pos -> virtpos) && backwards == YES)
               last_back = prev;
          cursize = pos -> current -> size - sizeof (struct free_item);
          if (cursize > bytes)
               cursize = bytes;
          if (cursize > 0)
               memcpy (pos -> curpos, from, cursize);
          pos -> virtpos += cursize;
          from += cursize;
          bytes -= cursize;
          }


/* : update the position info */
     pos -> curpos += cursize;
     pos -> end = pos -> curpos + pos -> current -> size -
          sizeof (struct free_item);
     return (last_back);
}
```

```
/* read information sequentially from unused storage */
void read_bytes (pos, to, bytes)
struct free_pos *pos;                   /* current position in free list */
char *to;                               /* where to read to */
int bytes;                              /* number of bytes to read */
{
     extern void memcpy ();             /* copy memory */

     int cursize;                       /* num bytes to copy this time */


/* : read as much as possible from the current unused item */
     cursize = pos -> end - pos -> curpos;
     if (cursize > bytes)
          cursize = bytes;
     if (cursize > 0)
          memcpy (to, pos -> curpos, cursize);
     pos -> curpos += cursize;
     pos -> virtpos += cursize;
     to += cursize;
     bytes -= cursize;


/* : keep writing to additional items until we're finished writing */
     while (bytes > 0) {
          pos -> current = pos -> current -> next;
          pos -> curpos = ((char *) pos -> current) +
               sizeof (struct free_item);
          cursize = pos -> current -> size - sizeof (struct free_item);
          if (cursize > bytes)
               cursize = bytes;
          if (cursize > 0)
               memcpy (to, pos -> curpos, cursize);
          pos -> virtpos += cursize;
          from += cursize;
          bytes -= cursize;
          }


/* : update the position info */
     pos -> curpos += cursize;
     pos -> end = pos -> curpos + pos -> current -> size -
          sizeof (struct free_item);
     return;
}
```

# Appendix H

# Baker's Re-rooting Algorithm

```
/* reroot an environment list */
NODE *reroot (leaf)
NODE *leaf;                              /* the new root node */
{
     NODE *root;                         /* the old root node */
     NODE *curnode;                      /* the current node in the list */
     NODE *next;                         /* the next node in the list */
     NODE *prev;                         /* the previous node in the list */
     NODE *value;                        /* the current lambda value */
     NODE *symbol;                       /* symbol the value is bound to */

     extern NODE *car ();                /* return the CAR of a list */
     extern NODE *set_car ();            /* modify the CAR of a list */
     extern NODE *cdr ();                /* return the CDR of a list */
     extern NODE *set_cdr ();            /* modify the CDR of a list */
     extern NODE *get_value ();          /* return value cell of a symbol */
     extern NODE *set_value ();          /* modify value cell of a symbol */


/* : traverse the path from leaf to root, reversing the list as you go */
     prev = NULL;
     curnode = leaf;
     while (curnode != NULL) {
          next = cdr (curnode);
          set_cdr (curnode, prev);       /* reverse list */
          prev = curnode;
          curnode = next;
          }
     root = prev;                        /* remember the old root */


/* : traverse the path from root to leaf, reversing as you go */
     prev = root;
     curnode = cdr (root);
     while (curnode != NULL) {
          symbol = car (car (curnode)); /* get symbol node from binding */
```

```
        value = cdr (car (curnode));  /* get value for symbol */
        set_cdr (car (curnode), get_value (symbol)); /* reverse binding */
        set_value (symbol, value);
        set_car (prev, car (curnode));     /* rotate one step */
        prev = curnode;
        curnode = cdr (curnode);
        }
    set_car (prev, NULL);              /* turn leaf into new root */
                                       /* (ie: complete the rotation) */

/* : return the old root node so the process can be reversed sometime */
    return (root);
}
```

# Bibliography


[Bake78a] Baker, Henry G. Jr., *List Processing in Real Time on a Serial Computer*, Communications of the ACM, Vol 21, No 4, Apr 1978, pp: 280-294

[Bake78b] Baker, Henry G. Jr., *Shallow Binding in Lisp 1.5*, Communications of the ACM, Vol 21, No 7, Jul 1978, pp: 565-569

[BartJ86] Bartley, David H., and Jensen, John C., *The Implementation of PC Scheme*, Proceedings of the 1986 ACM Conference on LISP and Functional Programming, 1986, pp: 86-93

[Betz86] Betz, David M., *XLISP - An Experimental Object-oriented Language*, Version 1.7. Jun 1986.

[BobrW73] Bobrow, Daniel G., and Wegbreit, Ben, *A Model and Stack Implementation of Multiple Environments*, Communications of the ACM, Vol 16, No 10, Oct 1973, pp: 591-603

[Broo84] Brooks, Rodney A., *Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware*, ACM Symposium on LISP and Functional Programming, Aug 1984, pp: 256-262

[Chen70] Cheney, C. J., *A Nonrecursive List Compacting Algorithm*, Communications of the ACM, Vol 13, No 11, Nov 1970, pp: 677-678

[Clar76] Clark, Douglas W., *An Efficient List-Moving Algorithm Using Constant Workspace*, Communications of the ACM, Vol 19, No 6, Jun 1967, pp: 352-354

[ClarG77] Clark, Douglas W. and Green, C. Cordell, *An Empirical Study of List Structure in Lisp*, Communications of the ACM, Vol 20, No 2, Sep 1977, pp: 78-87

[ClarG78] Clark, Douglas W., and Green, C. Cordell, *A Note on Shared List Structure in LISP*, Information Processing Letters, Vol 7, No 6, Oct 1978, pp: 312-314

[Cohe81] Cohen, Jaques, *Garbage Collection of Linked Data Structures*, ACM Computing Surveys, Vol 13, No 3, Sep 1981, pp 341-367

[CoheN83] Cohen, Jaques and Nicolau, Alexandru, *Comparison of Compacting Algorithms for Garbage Collection*, ACM Transactions on Programming Languages and Systems, Vol 5, No 4, Oct 1983, pp: 532-553

[Cour88] Courts, Robert, *Improving Locality of Reference in a Garbage Collecting Memory Management System*, Communications of the ACM, Vol 31, No 9, Sep 1988, pp: 1128-1138

[Danv87] Danvy, Olivier, *Memory Allocation and Higher Order Functions*, Proceedings of the SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques, June 1987 pp:241-252

[DeutB76] Deutsch, L. Peter and Bobrow, Daniel G., *An Efficient Incremental Automatic Garbage Collector*, Communications of the ACM, Vol 19, No 9, Sep 1976, pp: 522-526

[DewaM77] Dewar, Robert B. K. and McCann A. P., *MACRO SPITBOL - a SNOBOL 4 Compiler*, Software - Practice and Experience, Vol 7, No 1, Jan 1977, pp. 95-113

[FitcN78] Fitch, J. P. and Norman, A. C., *A Note on Compacting Garbage Collection*, The Computer Journal, Vol 21, No 1, Feb 1978, pp: 31-34

[HaddW67] Haddon, B. K. and Waite W. M., *A Compactiong Procedure for Variable-Length Storage Elements*, The Computer Journal, Vol 10, August 1967, pp: 162-165

[Hans69] Hansen, Wilfred J., *Compact List Representation: Definition, Garbage Collection, and System Implementation*, Communications of the ACM, Vol 12, No 9, Sep 1969, pp: 499-507

[HaynF84] Haynes, Christopher T., Friedman, Daniel P., *Engines Build Process Abstractions*, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Aug 1984, pp: 18-24

[HaynFW84] Haynes, Christopher T., Friedman, Daniel P., and Want, Mitchell, *Continuations and Coroutines*, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Aug 1984, pp: 293-298

[HollSSB80] Holloway, Jack, Steele, Guy L. Jr., Sussman, Gerald Jay, and Bell, Adam, *The SCHEME-79 Chip*, AI Memo No 559, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., Jan 1980

[Jonk79] Jonkers, H. B. M., *A Fast Garbage Compaction Algorithm*, Information Processing Letters, Vol 9, No 1, Jul 1979, pp: 26-30

[Knut73] Knuth, Donald E., *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, Addison - Wesley, Reading, Mass., 1973

[LangD87] Lang, Bernard and Dupont, Francis, *Incremental Incrementally Compacting Garbage Collection*, Proceedings of the SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques, June 1987 pp:253-263

[LiebH83] Lieberman, Henry and Hewitt, Carl, *A Real-Time Garbage Collector Based on the Lifetimes of Objects*, Communications of the ACM, Vol 26, No 6, Jun 1983, pp: 419-429

[LiH86] Li, Kai and Hudak, Paul, *A New List Compaction Method*, Software - Practice and Experience, Vol 16, No 2, Feb 1986, pp: 145-163

[McDe80] McDermott, Drew, *An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-Scoped LISP*, Conference Record of the 1980 LISP Conference, ACM, Aug 1980, pp: 154-162

[Moon78] Moon, David A., *The MACLISP Reference Manual*, Laboratory for Computer Science, M.I.T., Cambridge, Mass. 1978

[Moon84] Moon, David A., *Garbage Collection in a Large Lisp System*, ACM Symposium on LISP and Functional Programming, Aug 1984, pp: 235-246

[Morr78] Morris, F. L., *A Time- and Space- Efficient Garbage Compaction Algorithm*, Communications of the ACM, Vol 21 No 8, Aug 1978, pp: 662-665

[SchoW67] Schore, H. and Waite, W. M., *An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures*, Communications of the ACM, Vol 10, No 8, Aug 1967, pp: 501-506

[Stal80] Stallman, Richard, *Phantom Stacks: If you look too hard they aren't there*, AI Memo No 556, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., Jul 1980

[Stee75] Steele, Guy L. Jr., *Multiprocessing Compactifying Garbage Collection*, Communications of the ACM, Vol 18, No 9, Sep 1975, pp: 495-508

[Stee77] Steele, Guy Lewis Jr., *Data Representations in PDP-10 MacLisp*, Proceedings of the 1977 MACSYMA User's Conference. NASA Scientific Technical Information Office, Washington, D.C. Jul, 1977

[TeraG78] Terashima, M. and Goto, E., *Genetic Order and Compactifying Garbage Collectors*, Information Processing Letters, Vol 7, No 1, Jan 1978, pp: 27-32

[Unga84] Ungar, David, *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*, ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference, Apr 1984, pp: 157-167

[Wand80] Wand, Mitchell, *Continuation-Based Multiprocessing*, Conference Record of the 1980 ACM LISP Conference, ACM, Aug 1980, pp: 19-28

[Wegb72a] Wegbreit, B., *A Generalised Compactifying Garbage Collector*, The Computer Journal, Vol 15, No 3, Aug 1972, pp: 204-208

[Wegb72b] Wegbreit, Ben, *A Space Efficient List Structure Tracing Algorithm*, IEEE Transactions on Computers, Vol C-21, No 9, Sep 1972, pp: 1009-1010

[Wein88] Weinstock, Charles B. and Wulf, William A., *Quick Fit: An Efficient algorithm for Heap Storage Allocation*, SIGPLAN Notices, Vol 23 No 10, Oct 1988, pp: 141-148

[Weiz63] Weizenbaum, J., *Symmetric List Processor*, Communications of the ACM, Vol 6, No 9, Sep 1963, pp: 524-544

[WiseF77] Wise, David S. and Friedman, Daniel P., *The One-Bit Reference Count*, BIT Vol 17, No 4, pp: 351-359