

# A Garbage-Collecting Associative Memory for Database Systems

Richard Alan Ross

S.B., Massachusetts Institute of Technology  
(1978)

Submitted to the Department of  
Electrical Engineering and Computer Science  
in Partial Fulfillment of the Requirements  
for the Degree of


MASTER OF SCIENCE

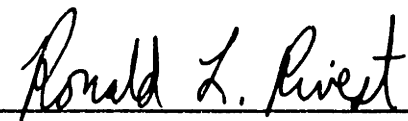
at the

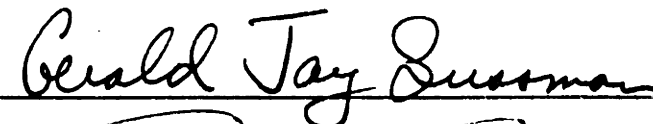
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1982

©Massachusetts Institute of Technology 1982

Signature of Author   
Department of Electrical Engineering and Computer Science  
May 20, 1982

Certified by   
Ronald L. Rivest  
Thesis Co-Supervisor

and   
Gerald J. Sussman  
Thesis Co-Supervisor

Accepted by   
Arthur C. Smith  
Chairman, Department Committee

ARCHIVES

MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

OCT 20 1982

LIBRARIES

# **A Garbage-Collecting Associative Memory for Database Systems**

by

Richard Alan Ross

Submitted to the Department of Electrical Engineering and Computer Science  
on May 20, 1982 in partial fulfillment of the requirements for the  
Degree of Master of Science in Computer Science.

## **Abstract**

This thesis presents an overview of a compacting garbage-collected associative memory, and gives details of its operational algorithms and data structures. Examples of its use are described, including an efficient implementation of LISP databases via property lists. A computational model of association objects is shown.

Garbage collection is the process of reclaiming inaccessible object names, and perhaps reassigning accessible object names. Garbage collection is performed to ensure that the system does not exhaust the set of object names, an important and finite resource. Associations are instances of a mapping of a list of objects to an object, referenced by means of their object names. Associations must be preserved by the garbage collector, presenting the problem: How may associations of objects be correctly and efficiently preserved in a system in which objects names are continually changing, due to garbage collection, and how may unnecessary associations be deleted from the associative memory?

In the past, data structures and algorithms which support both garbage collection and associative mappings have been seen as mutually exclusive. The main contribution of this thesis is to give techniques which implement a quick associative lookup, and properly modify the associative memory to reflect changes in object names due to garbage collection. A novel data structure which supports the cleanup of the associative memory, the dependency thread, is presented.

Thesis Co-Supervisor: Ronald L. Rivest  
Title: Associate Professor of Computer Science and Engineering

Thesis Co-Supervisor: Gerald J. Sussman  
Title: Associate Professor of Electrical Engineering

Keywords: Garbage Collection, Associative Memories, Database Systems, Property Lists, LISP, Hashing, Algorithms.

## Acknowledgments

I wish to thank my thesis advisors, Ron Rivest and Gerry Sussman, for their intellectual guidance and support, as well as for their substantial contributions to this work. I am indebted to Ron for the overall presentation, structure, and abstract framework herein. Gerry helped me clean up the algorithms, suggested the evolutionary presentation of Chapter 4, and originally described the problem to me. I was lucky enough to have two enthusiastic advisors, and I hope that this oeuvre pleases them.

Thanks to various folks around the Lab for Computer Science and the Artificial Intelligence Lab for sundry support, which included technical assistance, such as introducing me to the LISP Machine, SCRIBE, and EMACS facilities; critiquing previous drafts of this thesis and providing useful comments; and, most importantly, creating a place of good cheer and friendly company, making these most pleasant environs in which to work. This jolly group includes: Sandeep, Stavros, Tom, John M., Hal, Mike, Christos, Albert, Steve R., Steve T. (in spirit), Alan, Ravi, Neil, Susan, John B., Danny, and others whom I've undoubtedly forgotten.

D, Phil, Leslie, Greg, and Robbin each indirectly contributed to this thesis by their lively personalities and an optimistic spirits, allowing the author to enjoy the work throughout.

Special thanks to PSS for being a truly wonderful friend in more ways than I can say; and to NJG, in ways that I can't say.

This research was carried out with the support of a research assistantship from the Department of Electrical Engineering and Computer Science. This work was created using the text editor ZMACS, an EMACS variant running on the LISP Machine, and produced via the text formatter SCRIBE. I wish to thank the nameless creators and maintenance people of these facilities, for making my tasks all the easier.

*to my parents, whose love and support have shown me reach*

# Table of Contents

<b>Acknowledgments</b>	<b>3</b>
<b>Table of Contents</b>	<b>5</b>
<b>List of Figures</b>	<b>7</b>
<b>1. Introduction</b>	<b>8</b>
1.1 Overview	8
1.2 Property Lists	9
1.3 Notation	11
<b>2. Objects, Garbage Collection Schemes and Associative Memories</b>	<b>12</b>
2.1 Objects	12
2.1.1 Notation	13
2.1.2 Accessibility	14
2.1.3 Integrity Constraints on the Object Name Space	14
2.2 Garbage Collection	15
2.2.1 Definition	15
2.2.2 Methods	17
2.3 Associative Memories	19
2.3.1 Definition	19
2.3.2 Comparisons with Other Definitions	19
2.3.3 Operations	20
2.3.4 Methods	21
2.3.5 Hashing	21
2.3.6 Uses	22
2.3.7 Databases and Associative Memories	22
<b>3. The Problem and Solution Overview</b>	<b>23</b>
3.1 The Problem	23
3.2 Solution Overview	24
3.3 Dependency Threads and the List of Supported Associations	26
<b>4. Technical Analysis</b>	<b>28</b>
4.1 MFY Algorithm	28
4.2 The Associative Memory Garbage Collector	29
4.2.1 Method One: Save All Associations	29
4.2.2 Method Two: Disallow Embedded Associations	30
4.2.3 Method Three: Multiple Passes over the Associations	31
4.2.4 Method Four: Dependency Threads for Unsupported Associations	32
4.3 Virtual Names and Two-Level Lookups	34
4.4 LISP Databases	37
<b>5. Data Structures and Algorithms</b>	<b>38</b>
5.1 Data Structures for the Associative Memory	38
5.2 Algorithms	41
5.2.1 Associative Search Functions	41
5.2.2 Garbage Collection Processing	42

<b>6. Associations as Objects</b>	<b>46</b>
6.1 Object Functions	46
6.2 Association Operations	46
6.3 Sequencing	47
6.4 Generalized Pairing Functions	47
6.5 Garbage Collection of Association Objects	48
<b>7. Future Work and Summary</b>	<b>50</b>
7.1 Future Work	50
7.2 Summary	50
<b>Appendix</b>	<b>52</b>
<b>References</b>	<b>64</b>

## List of Figures

<b>Figure 2-1: Similar Objects</b>	13
<b>Figure 2-2: The Object Name Space</b>	16
<b>Figure 2-3: The Effects of Garbage Collection</b>	16
<b>Figure 3-1: Embedded Associations</b>	25
<b>Figure 3-2: The Depth of an Association</b>	25
<b>Figure 4-1: Association Space as a Sparse Matrix</b>	35
<b>Figure 5-1: The Data Structures for the Associative Memory</b>	39
<b>Figure 5-2: The Data Structures During Garbage Collection</b>	40
<b>Figure 7-1: CDR-Coding</b>	50

# 1. Introduction

In this paper we present the design, algorithms and data structures for an associative memory which can operate within a garbage collecting environment. To the best of our knowledge, this is the first time that an abstract associations which may be efficiently garbage-collected has been designed.

It is assumed that the reader is familiar with the programming language LISP. Also, some familiarity with garbage collection, associative memories, and database systems is assumed, although these concepts are briefly reviewed, in abstract versions, in Chapter 2.

Chapter 3 presents the problem with which this thesis is concerned: how may associations over abstract object names, which are reassigned by the garbage collector, be maintained by an associative memory? Chapter 3 also briefly describes the techniques which solve the problem.

Chapter 4 presents the solution in an evolutionary framework, and compares it with other methods which might be used to garbage-collect an associative memory. Chapter 5 gives a detailed presentation of the data structures and algorithms used in one specific implementation of our solution, describing how LISP property lists may be efficiently garbage-collected.

Chapter 6 examines the consequences of treating associations as first-class objects, and discusses the relevant issues, including the method of garbage-collection for the augmented object name space. Chapter 7 summarizes the results, and suggests areas of future research.

## 1.1 Overview

Garbage collection is the process of reclaiming inaccessible object names, and perhaps reassigning accessible object names. Garbage collection is performed to ensure that the system does not exhaust the set of object names, an important and (usually) finite resource.

Associations are instances of a mapping of a list of objects and constants to an object or a constant, with the objects referenced by means of their object names. Associations must be preserved by the garbage collector. This is the problem: How may associations of objects be correctly and efficiently preserved in a system in which objects names are continually changing, due to garbage collection, and how may unnecessary associations be deleted from the associative memory?

This thesis presents algorithms which implement a quick associative lookup, and modify the associative memory to reflect changes in object names due to garbage collection. A novel data structure



which supports the cleanup of the associative memory, the dependency thread, is presented.

## 1.2 Property Lists

The LISP property list facility supports the maintenance of database systems in LISP. This facility allows for the creation and subsequent access of ordered triples of the form "(key1,key2,result)." The element "key1" is known as the *object* of the association; the element *key2* is known as the *property*; the last is called the *value*. Property lists implement the creation of abstract associations in LISP through the use of the functions *put*, *get*, and *rem*; these procedures create (and modify), retrieve and delete associations, respectively.

The major problem with traditional implementations of property lists is that they are not properly garbage-collected. For example, if the value *C* is associated with the pair of keys *A* and *B*, *C* should be reclaimed by the garbage collector when *B* is, as the association is inaccessible. However, traditional garbage collectors retain *C* in this case (as well as *B*). These structures may not be used by the programmer, as they may not be accessed; however, these same structures, along with the (potentially vast) space they utilize, are needlessly kept by the garbage collector. This thesis presents a solution to this problem.

Property lists have traditionally been implemented as linear lists of ordered pairs of the form "(property,value)," called the *property list* for the object. An example of a property list for the object "apple" is:

```
((taste sweet) (weight 3.5) (size 2.2)).
```

As a value for the property "color" is added to the property list, it becomes:

```
((color green) (taste sweet) (weight 3.5) (size 2.2)).
```

One obvious problem in this traditional implementation is that when searching for a property of an object, the property list is searched linearly. This is a slow and inefficient search technique, except for a system in which there are only a few properties for each object.

As stated above, if the property key (first component) for an entry (property-value pair) in an object's property list is inaccessible, the entry will not be reclaimed by the garbage collector; as well, the value for that property (the second component of the entry) will not be reclaimed. For example, consider the following lines of LISP code which produce a property list for a symbol with a single property:

```

(setq list1 (cons 'b 'c))
(setq list2 (cons 'x 'y))
(put 'banana list2 list1)           ;its (b.c) prop is (x.y)
(get 'banana list1)                 ;get prop just installed
      ----> (x . y)
(setq list1 nil)                     ;make old (b.c) inaccessible
(get 'banana (cons 'b 'c))           ;can't get at prop
      ----> ()
(rem 'banana (cons 'b 'c))           ;and can't remove it
      ----> ()

```

If we could examine the property list for `banana` at this point, it would be

```
(((b . c) (x . y))),
```

showing that the property was not removed from the property list. Also, invoking the LISP garbage collector would not modify the structure of this property list.

More explicitly, in the above example, the structure whose `car` is `b` and whose `cdr` is `c` is `list1`. As the value of `list1` is changed by `setq`, the unique structure associated with `(b . c)` becomes inaccessible: there is no way that the system can recover use of it, so a programmer may not use it as an input to a function. A new "`(cons b c)`" invocation creates a *new* LISP conscell, different from the one that had been `list1` (*equal* but not *eq*).

However, the property list for the symbol `banana` contains a pointer to the original `(b . c)` structure, so it will not be reclaimed by the garbage collector. Similarly, the list structure `list2` that is `banana`'s value for the property will be saved unnecessarily. Note that this entry of the property list may neither ever be retrieved from the property list nor removed, as there is no way for the system to specify the desired property, but that the entry and all associated data structures (for the property and the value) are uselessly retained and are never garbage-collected. (The garbage collector is "conservative" in the sense that it will never garbage-collect an object that may be accessible; however, in the above case, the conservative garbage collector is inoperative.)

Property lists are apparently anathema to garbage collectors, as they prevent the latter from fully accomplishing their ends. This thesis presents an implementation of a general property list facility, however, which allows quick access time, allows associations over any number of keys, and may be fully garbage-collected.

The problem is presented within a generalized framework in this thesis; the solution is applicable to problems of garbage collection regarding capability-based protection systems, ACTOR message-passing mechanisms, hash tables (in which the entries depend on one another), as well as property lists.

Within the general framework, a conscell is an instance of an *object* (not the same as a LISP object with properties; see below), with its virtual address as its *object name*; *car* and *cdr* are the two *components* for this object type, and are also used as *selectors* for retrieving the object's internal components; symbols are instances of *constants*; property lists are a specific type of *associative memory*.

The problems described and solved in this thesis are not peculiar to traditional implementations of property lists. The design of an efficient, garbage-collecting associative memory is presented in general. It might be useful for the reader, however, to keep the concrete example of the LISP property list mechanism in mind while thinking about the abstract associative memory algorithms presented herein.

### 1.3 Notation

In this thesis, *italics* are used for the names of functions, program examples, emphasis, and definitions of terms. **Boldface** type is reserved for names of objects, component names, and names of specific instances of data structures.

The special LISP entities which have properties, the "object", will not be explicitly discussed further. Heretofore, all uses of the word "object" refer to the term as the basic computation structure of the model presented in the next chapter.

## 2. Objects, Garbage Collection Schemes and Associative Memories

The object model of computation is introduced. This model is used as a descriptive basis of garbage collection and associative memories.

### 2.1 Objects

In this thesis, the basic computation structure is the *object*. An object is an abstract entity composed of *components*. Each component of an object is either a system constant (like a number or a sequence of symbols), or is a *reference* (also called a *pointer*) to an object (described below).

Each object has a *name* by which it is referenced. That is, if one component of an object A is the name of object B, we say that A *references* or *contains a pointer to* object B. Examples of objects are LISP conscells, graph vertices, graph edges, processes, message-passing ACTORS, or virtual devices.

Objects are initially created by invocations of procedures known as *constructors*. Constructors retrieve an unused object name from the *freespace* (described below), form an object with the correct initial components, and return the object name for the newly created object. (This used object name is removed from the freespace to maintain system integrity constraints.)

Components are retrieved from objects by use of procedures called *selectors*. Let  $\Omega$  be the set of all object names. (We usually assume that  $\Omega$  is finite, as it supported by some real system resource.) Let  $D$  be the domain of  $\Omega$  unioned with the set of constants. A selector function  $\sigma_i$  is a mapping from an object name (chosen from the appropriate domain  $\Omega_i$ ) to an element of  $D$ :

$$\begin{aligned} D &= \Omega \cup \{\text{constants}\}; \\ \Omega_i &\subseteq \Omega; \\ \sigma_i: \Omega_i &\rightarrow D. \end{aligned}$$

An underlying *system support mechanism* supports the mappings of  $\Omega_i$  to  $D$  defined by various selectors.

*Mutators* modify the internal components of objects. The effect of a mutator  $\mu_j$  is reflected in subsequent selector invocations: A component of an object, retrieved by means of a selector invoked with the object name and component specified, may be different before and after the use of a mutator on that object:

$$\begin{aligned} \sigma_i(A) &\rightarrow 7 \\ \mu_j(A, 5) & \\ \sigma_i(A) &\rightarrow 5 \end{aligned}$$

Each object has a *type* which specifies which selectors and mutators are appropriate to the object. An object is given its type when it is originally constructed.

It is seen that objects form a directed graph structure with objects and constants as vertices and object pointers as edges. There is usually one distinguished object, the *root-object*  $r_0$ , from which all other objects may be reached by means of traversing the object pointers.

Two apparently distinct objects are the *identical* (*eq* in LISP) if they have the same object name, so references to them resolve to the same object; these objects are aliases for one another. Two objects for which the application of selectors yield the same constants are *equivalent* (*equal* in LISP terminology); these objects contain the same information, from a functional viewpoint. Two objects are *isomorphic* (no corresponding predicate in LISP) if and only if the directed graph structures over the labelled digraphs formed by these objects can be shown to have a one-to-one correspondence in the vertices (object names and constants) and edges (references); these objects are copies of one another. (Please see Figure 2-1 for a comparison of these relations between objects.)

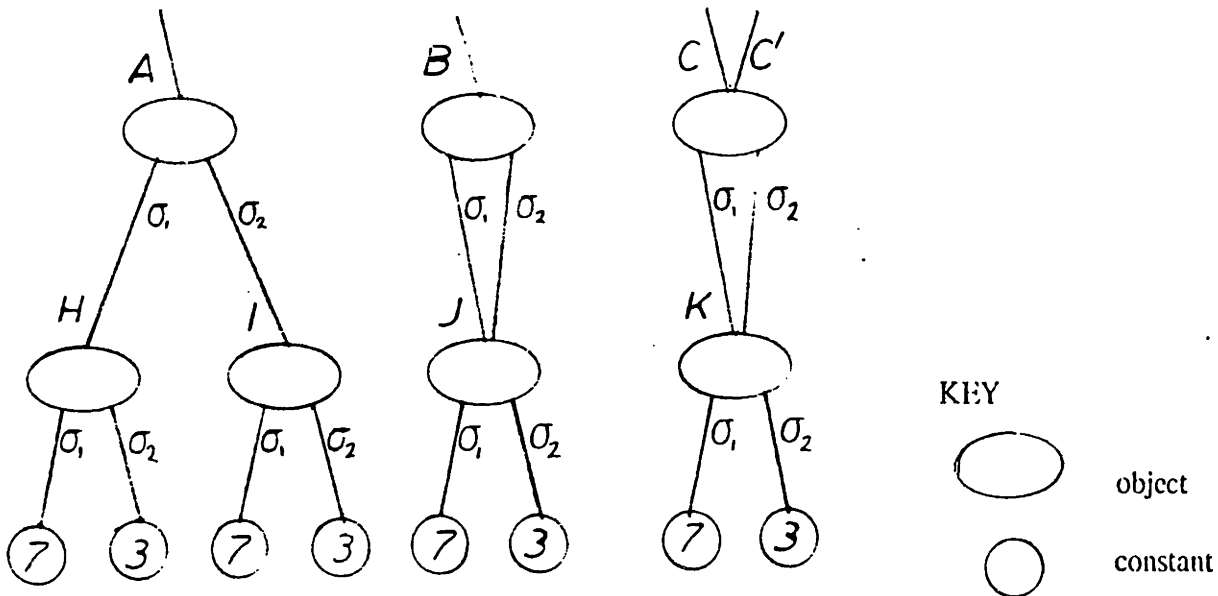


Figure 2-1: Similar Objects

A, B, C, and C' are equivalent;  
 B, C and C' are isomorphic; C and C' are identical

### 2.1.1 Notation

The object name (reference) or constant which is contained in a component of an object is

designated in this thesis by using the component name as a suffix; that is, the component name is informally used as a selector to retrieve the component. For example, consider a tree-like object with components `leftson`, `rightson`, and `sibling`. An object called `node1` would have as a component `node1.rightson`; this would be the same entity (constant or object name) as would be returned by the selector *select-rightson*(`node1`).

A component name suffix used on the left hand side of an assignment statement is an abbreviation for a mutator call. That is, the statement `node1.sibling := node15` is shorthand for the mutator call *mutate-sibling*(`node1,node15`).

### 2.1.2 Accessibility

An object `B` is *pointer-accessible* from an object `A` if and only if object `B` is: 1) the object `A` itself; 2) referenced by `A`; or 3) referenced by an object `C` that is pointer-accessible from `A`. Equivalent to this recursive definition, an object `B` is pointer-accessible from `A` if and only if it may be reached from object `A` by some finite number of successive invocations of selector functions. An object is *root-accessible* if it is pointer-accessible from the unique root-object.

An object `A` that had previously been root-accessible may become root-inaccessible as mutators are applied to those objects which had referred to object `A`. (In this case, `A` would become a member of the *garbage space*, defined below.) A root-inaccessible object name may become root-accessible only if it had been an element of the *free space* (defined below) and was put into service by the constructor procedure as a new object was created.

### 2.1.3 Integrity Constraints on the Object Name Space

The object name space,  $\Omega$  may be partitioned into three subsets `R`, `F`, and `G`. `R` is defined to be the set of all root-accessible objects. An object `A` is an element of the set `R` if `A` is pointer-accessible from the root object `r0`.

The second subset of  $\Omega$ , `F`, is defined to be the set of all *free* object names. These free object names are those which are not in use (not in `R`), and may be obtained by the constructor functions, as needed, to be used as names for objects to be created. The space `F` must be explicitly maintained (by the underlying system support mechanism) for this purpose.

Two global operations that modify `F` must be allowed: there must be an *extract-element-from-F* command, to be used by the constructor functions, which removes an object name from `F` upon demand,

and returns that removed object name. Similarly, there must be an *install-element-into-F* command, used by the garbage collector (described below), which adds an object name to  $F$ . Commonly,  $F$  is maintained as a linked list structure, so the extraction and installation of object names in  $F$  are similar to the maintenance operations for a stack or queue.

The last subset of  $\Omega$ ,  $G$ , the *garbage space*, is the set of used object names which are not yet reclaimed into the freespace. These are object names which had been in  $R$ , but are no longer (due to the invocations of various mutators), and have yet to be returned to the freespace  $F$ . As these object names may neither be obtained by the constructor functions nor are root-accessible, they are useless to the system. These objects names are garbage, and should be returned to the freespace for later use. (Please see Figure 2-2 for a depiction of these subsets of  $\Omega$ .)

The two integrity constraints on the system are that the subsets  $R$ ,  $F$ , and  $G$  are mutually exclusive and collectively exhaustive on  $\Omega$ :

$$\begin{aligned} R \cap F &= R \cap G = F \cap G = \emptyset, \\ R \cup F \cup G &= \Omega. \end{aligned}$$

Note that this necessitates that each new object name placed into  $R$  by use of a constructor is immediately removed from  $F$ .

## 2.2 Garbage Collection

### 2.2.1 Definition

Garbage collection is the process of reclaiming (removing from  $G$  and installing back into  $F$ ) object names that are neither root-accessible nor in the freespace, i.e., that are part of the garbage space; the garbage collector may be viewed as the object name space manager. As the object name space is finite (by assumption), garbage collection is used to ensure that a system does not run out of object names for new objects. (A compacting garbage collector may also speed up the system, for it produces greater locality of reference for object names by reallocating them to be contiguous in  $\Omega$ .)

Garbage collection must *free* (or *reclaim*) the object names of root-inaccessible objects, returning these object names to the freespace. Similarly, it must *protect* those object names which are root-accessible. The following are the integrity constraints that a garbage-collected system must satisfy, in addition to the two listed above (primed spaces are those after the garbage collection has been accomplished, and  $|S|$  is the size of the set  $S$ ):

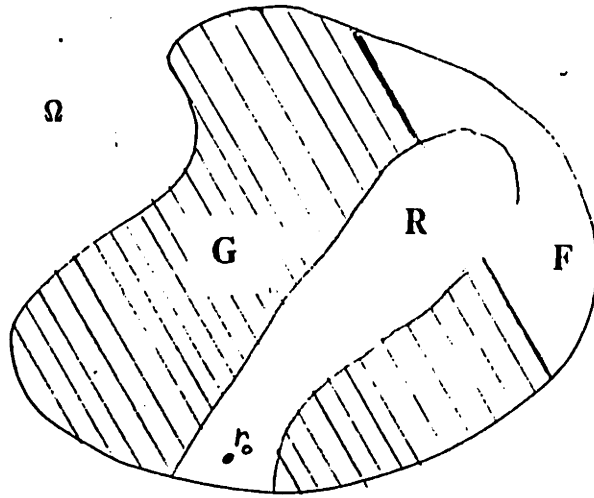


Figure 2-2: The Object Name Space

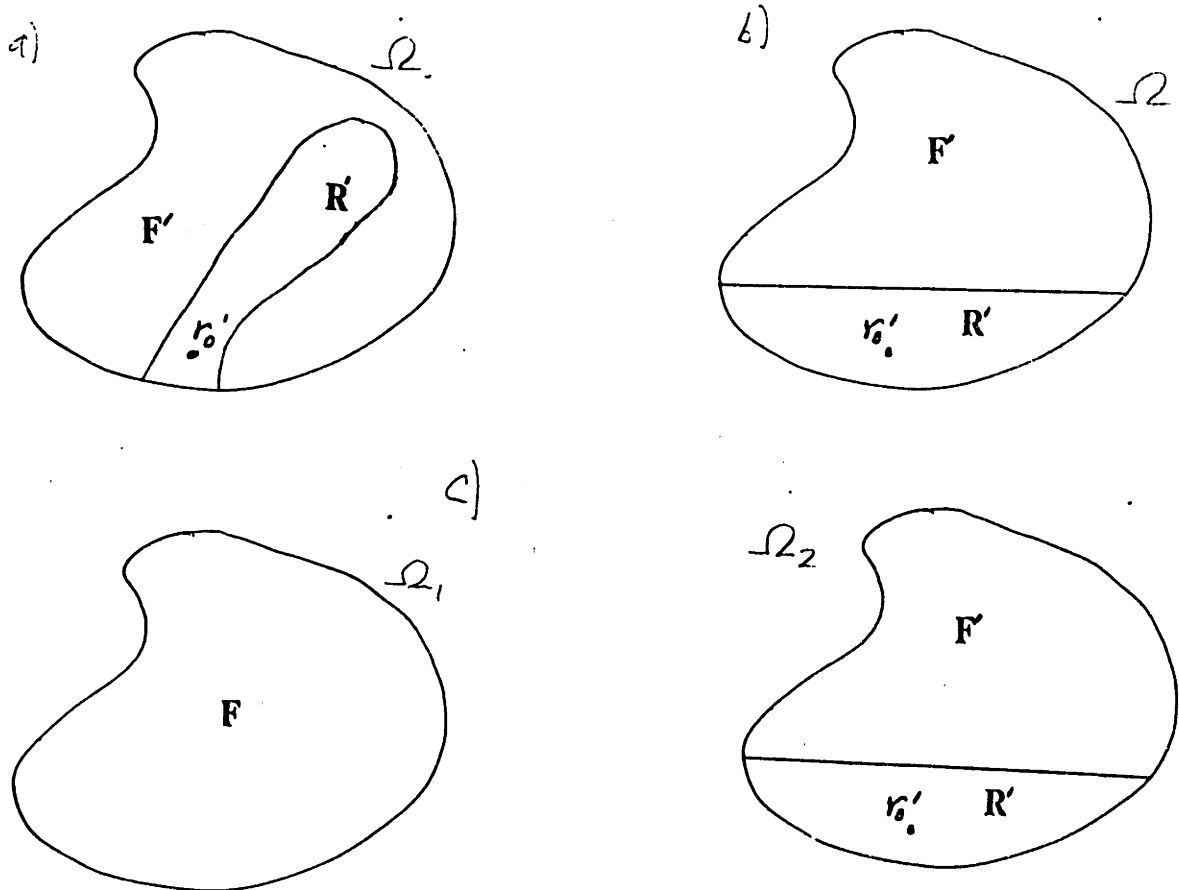


Figure 2-3: The Effects of Garbage Collection

a) simple sweep; b) compacting sweep; c) copying collector



1.  $|G'| < |G|$  (some garbage is collected);
2.  $|F'| = |F| + (|G| - |G'|)$  (all garbage-collected resources are returned to the freespace);
3.  $R'$  is isomorphic to  $R$  (accessible object names are protected by the garbage collector).

This isomorphism is as defined in section 2.1. Isomorphism is used to ensure that the old and new root-based structures are computationally equivalent, and that all programs (which are object structures, specially interpreted by the system) work correctly. A garbage collector guarantees isomorphism by copying the graph structure of the root object; all compacting garbage collectors, for example, must copy and reset the pointers within the objects to maintain structure and ensure the integrity constraints.

Usually, garbage collection is invoked as needed, when  $F$  is very nearly exhausted. "Real-time" garbage collectors are those that are called every time an object is created; they work incrementally, doing a little of the garbage collection at a time [2, 22]. Concurrent garbage collectors, requiring an extra (perhaps virtual) processor, work simultaneously with the main process that is manipulating objects [35, 8].

One important note: no garbage collector should need an auxiliary stack for control, as garbage collection occurs because resources are scarce. A garbage collector which needed an unbounded control stack could find itself easily deadlocked.

Garbage collectors often do, however, require a fixed amount of storage per object name that is garbage-collected. For example, mark and sweep garbage collectors use a mark bit to note that an object name has been tagged (and perhaps reassigned); a depth-first search algorithm employed to find the accessible objects may build a stack within the structure of root-accessible objects themselves (but not use extra resources), as in the Deutsch-Schorr-Waite marking algorithm [5].

### 2.2.2 Methods

Common methods for garbage collection include reference counts, mark and sweep algorithms, and copying algorithms [19]. (Also, hybrid systems have been developed which incorporate features from different types of garbage collectors [7].) Briefly, reference counts indicate, for each object, how many other objects in the object space point to it. When the reference count for an object becomes zero, it is root-inaccessible, and the object name is returned to the freespace. Reference count schemes are unable to garbage collect circular structures, and therefore do not accomplish full garbage collection.

Mark and sweep algorithms proceed in two phases. The mark phase tags (marks as protected) all

root-accessible items (via a depth first search starting from the root-node, for example). The sweep phase returns all untagged object names to the freespace; it may *reassign* the object names of all tagged items so as to be contiguous (e.g., if the resource employed -- the object name space -- is main memory, this process of *compaction* moves all used storage into one block of memory; the procedure of reassigning object names is called *relocation*). This method suffers in that the object name space must be scanned during the sweep phase. This time is great for large object name spaces.

(Often, the object name space is embedded in a subset of a real system resource, such as primary memory, secondary memory, microprocessors, or address space. In this case, the resources as object names are used by the system to distinguish between objects. That is, *the (system) name for an object is the resource that the object is utilizing*. For example, the system name for a conscell may be "the conscell residing at location 26"; the system name for a virtual process object may be "the object running on microprocessor number 7." This phenomenon is a result of past and present programming practices; in those systems in which this phenomenon does occur, the object name space  $\Omega$  may not be the complete resource space, but only a subset of it. For example, for LISP conscell objects, the object names may just be the even numbered virtual addresses in a system; or the object name space may be a set of starting locations for LISP array objects. Compacting garbage collectors force the real system resources being used by the object name space to be contiguously allocated. In this way, there is room for large objects, which require much real resources; also, the "working set" of the system resources in a paged environment is kept relatively small.)

Copying garbage collectors trace down the root-accessible objects and reassign their object names to be contiguous in a new object name space as they are found. Usually, twice as much space is needed as in other methods (i.e., space where object names are,  $\Omega_1$ , and a new object name space,  $\Omega_2$ , in which object names will initially reside contiguously), but all garbage collection and compaction is performed without a complete scan of the object name space. In some sense, copying garbage collectors perform mark and compacting sweep operations at the same time, and use the newly compacted objects as a breadth-first search queue for marking.

The results of a simple sweep, a compacting sweep, and copying garbage collection are pictured in Figure 2-3. An excellent survey article on garbage collectors has recently been published by Cohen [5], and the recent book by Standish [34] includes much on garbage collection data structures and techniques. The interested reader is referred to these works for further description of garbage collection methods.

As non-compacting garbage collectors suffer the indicated problems, it will be assumed from herein

that we are given a garbage collector for the object name space which does reassign the object names that it saves.

## 2.3 Associative Memories

As mentioned in the Introduction, the property list facility of LISP is an example of an associative memory. It allows the user to establish connections between object (conscells) of the system. This associative memory may be used as the basis for frame representation languages [30] and data-directed dispatch mechanisms.

The associative memories described below are generalizations of the LISP property list facility. Those below allow any object or constant in the domain to be used as a key for association.

### 2.3.1 Definition

An associative memory is a device which maps an a list of object names or constants, the *keys*, into an object name or constant, the *value*:

associative memory:  $D^k \rightarrow D$ ,  $k$  constant.

*Associations* are special objects which help support the associative mappings. Associations reside in the association space, and may not be referenced by or as first class objects. (In a later chapter, this restriction is relaxed, and associations are placed in the object space and given first class status.) Associations, as objects, have names; the name of an association is its *association-id*, and is an element of  $\mathcal{A}$ , the set of all association-id's.

### 2.3.2 Comparisons with Other Definitions

In its most general definition, an associative memory implements many functions. If each key may be specified as being within a certain range, then *range queries* are supported. (This assumes that a total order on the keys is possible.) If only ii(some) of the keys need be given to retrieve the values of acceptable associations, then *partial match queries* are supported. (The report by Rivest [29] covers the various types of key specification in more detail. The book by Kohonen [21] discusses various metrics, used for best match queries, over the space of the keys; as well, it gives examples of many types of associative memories.) In this thesis, only associative memories which support *exact match queries* are considered; each key for an association must be fully specified as an object name or a constant, and each specified key must match exactly (be identical objects to) the association object's key components.

In other associative systems, only constants may be used as the keys for associations. In this case, an

associative memory is a mapping:

$$C = \{\text{constants}\}$$

associative memory':  $C^k \rightarrow C$ ,  $k$  constant.

For an associative memory of this sort (which is, indeed, the most usual type for all but the artificial intelligence community), the keys may be totally ordered, so support of partial match and range queries is natural. This thesis, however, allows key specification as either object name or constant.

To reiterate the comparisons, the model presented is more restrictive than the usual associative memory, in the sense that only exact match queries are supported; but the model is also more general, as any arbitrary object name or constant may be a key for association. Note that range queries make little sense in the generalized framework, as no metric or total order can be easily imposed on the object *name* space; the names have no relation to the object they allow to be referenced. (Also, these names are reassigned by the garbage collector, so any metric would have to dynamically reflect these changes.) Partial match queries are a reasonable feature within our model, but we have found no way to implement them efficiently.

### 2.3.3 Operations

An associative memory must be able to perform three operations:

1. create an association between an ordered list of keys and a given value, or modify a previous association (*put*);
2. retrieve the associated value, given the keys (*get*);
3. delete the association for a given list of keys (*remove*).

We shall assume that the length of the key list for any one associative memory system is fixed. For a particular system, we refer to this constant as  $k$ , the *dimension* of the associations. Therefore, *put* is a function of  $k + 1$  object names, while *get* and *remove* take  $k$  inputs. The dimension  $k$  is assumed to be greater than one, else an associative memory may be implemented as a selector for objects (see below); the value of two for  $k$  is the usual case for LISP property list mechanisms, and a  $k$  value of three might be useful in many instances (including a context for the association as a key, for example).

For  $k = 1$ , the associative memory is trivially easy to implement: each object, whose name would be the single key for an association, could have the association value as one of its components! Also, note that the dimension for any associative memory may always be viewed as being equal to one, since each list of  $k$  keys may be interpreted as one large key, an element of the Cartesian product  $D^k$ , where  $D$  is, as

above, the domain for the keys,  $D = \Omega \cup \{\text{constants}\}$ . However, in this case, the object names, and the object name space, are huge, as they are composed of concatenations of smaller object names drawn from a smaller space.

#### 2.3.4 Methods

Associative memories are often hard-wired devices which perform an associative search ("lookup") for a value given a set of keys. The methods for associative searches are well-established. Common schemes include the use of linked lists; binary trees or sorted lists [20], 2-3 trees [43], B-trees [17], tries [13] or discrimination nets [10]; and hashing with various collision resolution schemes [20, 1]. The first method gives linear time (in the number of associations) response on the average (and in worst-case, also); the next group of methods all have varying degrees of logarithmic average time response; hashing has constant response time on the average, given a sufficiently large hash table.

#### 2.3.5 Hashing

As stated above, it is well known that associative searches based upon hashing perform quite well on the average. We review some of the common definitions associated with the implementation of hashing employed in our system:

A *hash function* maps its inputs, the keys for association, into a unique integer, the *hash number*; this hash number is interpreted as the index into an array, the *hash table*, which stores the associations. If two different sets of input keys map into the hash number via the hash function, *collision* occurs. To handle this common occurrence, each element of the hash table, rather than being an association itself, points to a *bucket* containing many associations (all those colliding on the same hash number). Each bucket is a *linked list* or *chain* of those associations within the bucket. This bucket is linearly searched when a specific association is to be retrieved.

It may appear that the worst case time for hashing (when all associations collide) is abysmally slow, and so that hashing is an unacceptable method for performing associative searches. However, recent work has shown that this is an extremely rare occurrence, as might be imagined. Indeed, the expected worst case search time, assuming equiprobable distribution over the keys, is better than logarithmic; the actual time is on the order of the inverse gamma ("factorial") function [15].

Current research continues on a description of effective hash functions. Sets of hash functions with effective hashing properties (keys are distributed over the hash numbers randomly yielding balanced buckets, within a high statistical measure), called "universal classes of hash functions," have recently been

described [6, 42, 40, 32, 28, 24].

### 2.3.6 Uses

Associative memories are useful for capability-based protection systems [9], ACTOR-like message passing systems [4, 2], object-based architectures, frame representation systems [26], and LISP property list mechanisms [31]. An associative memory is the underlying support for database systems (as mentioned below).

The reference *Sorting and Searching* by Knuth [20] is the classic work on the subject of associative searches. Also, an article by Feldman [11] presents many possible uses of a full-functioned associative memory.

### 2.3.7 Databases and Associative Memories

The basic device which supports a database system is an associative memory [20, 41]. While the *logical representation* of a database system may be in terms of functions, relations, or records, its *system level (operational) representation* will be based upon an associative memory. Hence, a well-designed associative memory is necessary for an efficient database system.

The property list facility of LISP is the usual mechanism with which LISP databases are maintained [31]. Large databases for artificial intelligence processing are usually supported by linear property lists, and the so increased efficiency in the LISP property list facility (as presented herein) may vastly improve these (usually monstrously slow) programs. As well, the ability to garbage-collect the association space saves system resources.

The use of hash functions and tables to support the LISP property list facility is well-known [3]. However, naive implementations of the LISP functions *put*, *get*, and *rem* using hashing cannot operate in the standard LISP garbage-collecting environment, or else the association functions are restricted to operating on (non-changing and un-garbage-collected) LISP constants (symbols) instead of objects (conscells).

## 3. The Problem and Solution Overview

### 3.1 The Problem

As shown in the Introduction, the property list facility of LISP may not be properly garbage-collected. Within the generalized object model of computation presented in this thesis, the major problem that we wish to address is: What is the design for an efficient associative memory that must work in an environment in which the object names are changed during each garbage collection period? That is, the problem is to design an associative memory that can handle the following type of question efficiently: "What is the value of the association between the object whose name is currently *obj-25* (which name had been different when the association was originally established), and *obj-446*?"

There are two different types of garbage collection involved in this problem:

1. An object name space garbage collector (*osgc*), similar to those garbage collectors described above, operates in the object name space, reclaims inaccessible object names and modifies the object names of accessible objects.
2. An associative memory space garbage collector (*amgc*) must modify the private memory of the associative memory to guarantee that all associations of lists of accessible key objects and the corresponding values are maintained after the garbage collection, and all associations in which one or more of the associated key objects are inaccessible are deleted.

Note that if a value to be kept is a member of the object name space, it must be explicitly saved (protected, and perhaps reassigned) in the object name space by a call from the *amgc* to the *osgc*. Furthermore, values (objects) that are accessible through legitimate embedded structures of association (*embedded associations*) must be maintained. For example, "*get(get(obj-1,obj-2),obj-4)*" must be allowed after the garbage collection process (assuming it was well-defined before), even if the object which is the value associated with the pair (*obj-1,obj-2*) were not directly root-accessible through the the object name space only. (Please see Figure 3-1.) Of course, purely circular structures which are inaccessible must be deleted from the associative memory.

Recall that, previously, an object could have been used by the system if it had been pointer-accessible from the unique root-object. That is, an object was root-accessible if it had been the result of a selector path which originated at the root.

Now, however, the term *accessible* has an enhanced meaning; an object is *accessible* if it is either: 1) root-accessible; 2) the value of an association of accessible objects; or 3) pointer-accessible from an



The Libraries  
Massachusetts Institute of Technology  
Cambridge, Massachusetts 02139

Institute Archives and Special Collections  
Room 14N-118  
(617) 253-5688

This is the most complete text of the  
thesis available. The following page(s)  
were not included in the copy of the  
thesis deposited in the Institute Archives  
by the author:

p. 24



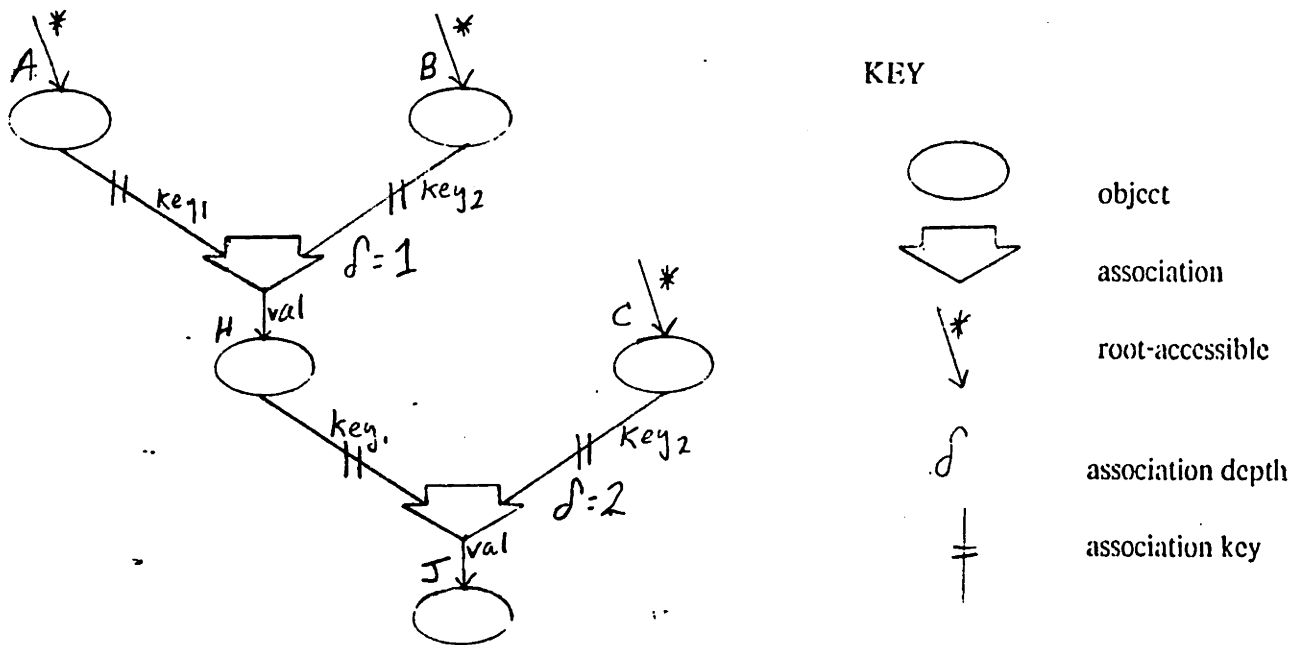


Figure 3-1: Embedded Associations

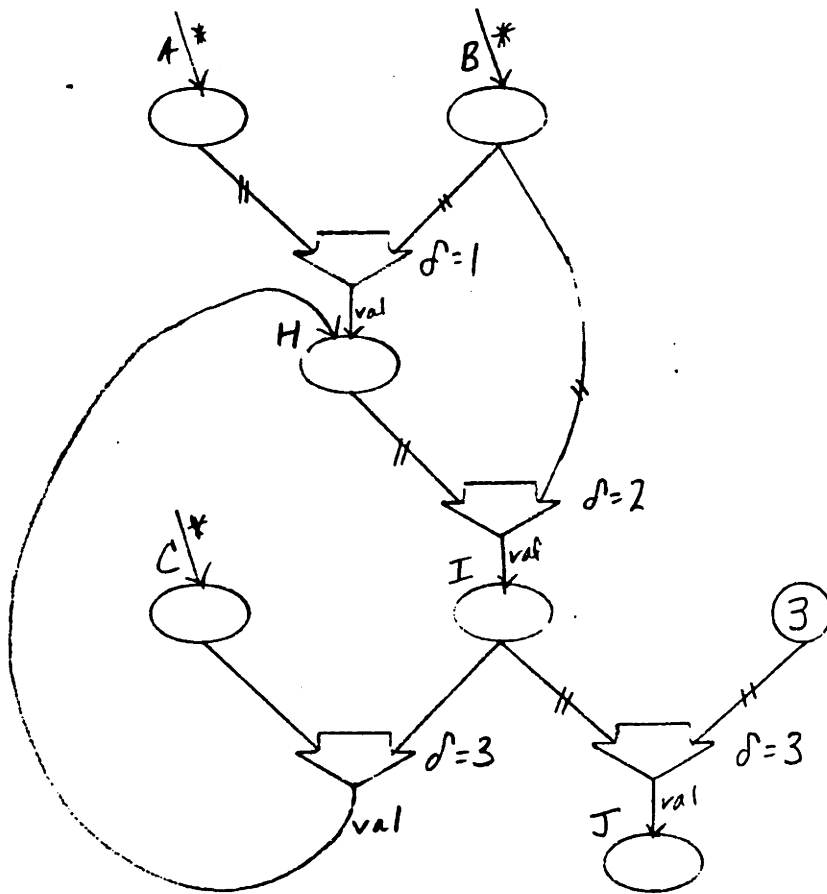


Figure 3-2: The Depth of an Association

For processing of a *put*, *get*, or *remove* request, the first level of the look-up, *virtual name resolution*, or *vnr*, maps each key object name to a unique "virtual name" via hashing with linked list collision resolution. The second level look-up, *association unification*, or *aun*, maps lists of virtual names to values (object names), again using hashing and linked list resolution.

The associative memory garbage collector first traces down the support structure of virtual names at the *vnr* level (which had been used to map object names to virtual names) and rehashes to find the new bucket for each root-accessible object. Objects that are not yet known to be accessible are allocated *dependency threads* to allow for the fact that they may be protected and reassigned in the future (as they may be the keys for some embedded association and the values of other associations, as explained above).

At the next stage of the *amgc*, associations at the *aun* level are copied (actually, put on the special global list, *supported-assoc-list*, to indicate that they are to be copied in the future) if they are *completely supported* by the object name space; that is, if all the key objects of the association are accessible. For an unsupported association, the association object is added to the dependency thread for each of its inaccessible key objects. If all of an association's key object names are ever reassigned by the *osgc*, and hence accessible, in the future, the association is re-created; else, it will be implicitly destroyed at the end of the garbage collection process.

The *amgc* delays installation of (perhaps initially unsupported) associations by placing them on the global list, to ensure that reassigned object names which are the values of associations do not cause other dependency threads to be immediately processed. Otherwise, a situation could occur in which reassigning a value may cause other values to be reassigned in a recursive manner, entailing the use of an auxiliary control stack. As shown in the next section, the space for the data structures which support this delayed installation technique is found in the remnants of the association objects; no extra resources are required.

### 3.3 Dependency Threads and the List of Supported Associations

The algorithms critically depend upon the maintenance of the special data structures, the dependency threads and the *supported-assoc-list*. These data structures are created within the remnants of the old associations, and hence require no auxiliary space for their support.

An object name *supports* an association if that association has the object name as one of its keys. In the same instance, we say that the association *depends upon* that object name.

During garbage collection, a dependency thread is allotted to each (potentially accessible) object which supports an association. The dependency thread of an object maintains a (threaded) list of all those associations which depend upon that object name (i.e., have that object name as a key). The associations themselves contain the thread pointers which are the constituent parts of the dependency thread. It is seen that the dependency thread for an object is fragmented, and is spread throughout the association-id space, and actually resides in all those associations which are on the thread. (Please see Figures 5-1 and 5-2 for a concrete example of the data structures used in an implementation of a garbage-collecting associative memory for the LISP property list facility.)

There is one list of associations which are completely supported, which is maintained during garbage collection, the **supported-assoc-list**. It also is maintained in a fragmented fashion, and is scattered amongst the internal components of the associations. When an association is discovered to be completely supported, with all its keys accessible in the object name space, the association is placed on the list. Explicitly, it is attached to the beginning or the end of the **supported-assoc-list** (for stack- or queue-like processing, respectively); as such, the parts of the **supported-assoc-list** reside in all the associations on the list. As associations are removed from the list, they are copied into the next (saved) version of the associative memory, and their value components are examined to detect more accessible objects.

It is seen that the special data structures borrow space from the associations themselves. As such, the dependency threads and the **supported-assoc-list** require only a small, constant amount of extra space for maintenance and processing.

During garbage collection, an association may undergo various metamorphoses. It may become part of many dependency threads (at most, one for each of its keys); it may find its key become accessible, one by one, and may eventually become part of the list of supported associations; finally, it is replicated for the new version of the associative memory.

## 4. Technical Analysis

The Minsky-Fenichel-Yochelson [25, 14] garbage collector is an efficient technique for garbage-collecting large object name spaces, and its mechanisms are presented in this chapter. The evolutionary development of an efficient algorithm for garbage-collection of associative memories is shown; also given are search algorithms for an associative memory which minimize rehashings due to garbage collection.

### 4.1 MFY Algorithm

The garbage collector for the object name space, previously installed in our system, uses the Minsky-Fenichel-Yochelson copying-compacting algorithm. Two copies of the object space are necessary; the *oldspace* is where objects to be saved reside, while the *newspace* is the place to which they are saved by the garbage collector. The oldspace and newspace change roles every garbage collection period; these spaces, as well as the global variable *freeptr* which points to the next free object name in the current newspace, must be initialized before the garbage collection processing occurs.

Objects are moved from the oldspace to the newspace as they are found to be accessible. As the newspace is scanned for newly saved accessible objects (and their pointers to other objects), it is used, in effect, as a breadth-first search queue. The *osgc-protect* procedure is called with a pointer to the highest level object to be protected (and reassigned); all objects pointer-accessible from this input object name are also protected, as is seen in the following "meta-programs":

```

proc osgc-init()          /*initialize spaces and freeptr */
begin
  initialize-spaces();      /* set up spaces */
  freeptr := beginning-of-newspace(); /* all space is clear */
end osgc-init.

proc osgc-protect(oldobj) returns pointer newobj
begin
  newobj := copy-old-to-new(oldobj); /* save new high level ptr */
  scanptr := newobj; /* start search at beginning */
  until all-objects-scanned-in-newspace(scanptr,freeptr) do /*space as queue*/
  begin
    next-obj := next-obj-in-newspace(scanptr); /* this obj already protected */
    scanptr := next-obj; /* update scanptr */
  /* find new accessible obj's: */
  for each component, compi in next-obj do /*save all nec'y parts */
    if not-constant(next-obj.compi)
      then next-obj.compi := copy-old-to-new(next-obj.compi);
    end;
  return(newobj);
end osgc-protect.

```

The garbage collector uses an auxiliary function which copies an object from the oldspace to the newspace, given a pointer to the object. This function returns the new assignment of the object name:

```

proc copy-old-to-new(oldptr) returns pointer newptr
begin
  oldobj := object-in-ospace(oldptr);
  if already-reassigned?(oldobj)
  then newptr := new-assignment(oldobj) /* point to copy in newspace */
  else begin /* or make a new copy */
    for each component, comp, in oldobj do
      copy-component(oldobj.comp, freeptr) /* copy all obj into newspace */
      note-reassigned(oldobj, freeptr); /* tell new assignment */
      newptr := freeptr; /* new object assignment */
      freeptr := next-free-obj-name(freeptr); /* update freeptr */
    end;
  return(newptr);
end copy-old-to-new.

```

(Note that the MFY algorithm may be modified to obtain a single source reachability algorithm that uses no extra storage (save output array), gives shortest paths, and operates in time  $O(|E|)$ . Iterative use of this algorithm leads to an  $O(|E| \cdot V)$  time algorithm for transitive closure with the indicated properties.)

## 4.2 The Associative Memory Garbage Collector

The basic design philosophy for the associative memory's garbage collector is to save all associations that are accessible through either the object space or through embedded associations via the associative memory. This is the basis for the first major problem to be solved.

To handle garbage collection of association cells in a system with embedded structures, one could:

1. copy all associations, whether accessible or not;
2. not allow embedded structures at all, or only allow embedded associations of bounded depth;
3. make multiple passes through the aun level hash table to find newly accessible associations;
4. make note of the objects upon whose accessibility an association depends, and save the association as the object becomes accessible.

Each of these options will be examined.

### 4.2.1 Method One: Save All Associations

The first method, the naive one of copying all associations, is the easiest to implement:

```

proc assoc-garbage-collect1 ()      /* copy all associations */
begin
  for each association, assoc, in associative memory do
    if not-marked-deleted?(assoc) /*not deleted by old remove? */
      then begin /* ok, save cell */
        osgc-protect(assoc.value);
        amgc-save(assoc);
      end;
    end
end assoc-garbage-collect1.

```

With this solution, all associations are saved, except those which had been deleted by means of a previous *remove* request. As an association is saved, its value object is protected by an *osgc-protect* call to the *osgc*; the association is updated (to reflect changes in its key components due to prior reassignments by the *osgc*) and re-installed in the association space, by an *amgc-save* invocation.

As even unsupported associations (those having inaccessible key objects) are saved, this method is wasteful of space, and really avoids the whole issue of garbage collection.

#### 4.2.2 Method Two: Disallow Embedded Associations

A slightly more sophisticated garbage collection scheme avoids the issue in a different manner, as it disallows embedded associations entirely:

```

proc assoc-garbage-collect2 ()      /* copy non-embedded assoc's */
begin
  osgc-protect(root-node); /*!!! mark all root-accessible objects */
  for each association, assoc, in associative memory do
    if not-marked-deleted?(assoc)
      then if assoc-supported?(assoc) /*!!! accessible from outside? */
        then begin
          osgc-protect(assoc.value);
          amgc-save(assoc);
        end;
      end
    end
end assoc-garbage-collect2.

```

Note that the object space garbage collector is initially called so that all root-accessible objects may be discovered; it is assumed that all spaces has previously been initialized by an *osgc-init* invocation.

In this method we save an association if an only if it is completely supported upon first examination; that is, an association is saved if *all* its keys are root-accessible. An auxilliary function tests if this condition is true:

```

proc assoc-supported?(assoc) returns boolean support-flag
begin
  support-flag := true;      /* default: supported */
  for each keyi in assoc do
    if not-osgc-protected?(assoc.keyi) /* each key saved outside? */
      then support-flag := false; /*if not, assoc not fully supported: */
    return(support-flag);
  end assoc-supported?.

```

This second method does not let us realize our design objective of allowing embedded associations of unbounded depth. It does achieve the garbage collection of purely circular structures (as these have inaccessible keys), which is an improvement over our first attempt. Note that we could allow limited embedded associations, say of bounded depth  $i$ , by making  $i$  passes of garbage collection. With each pass, a deeper level of embedded associations would be saved, since the values protected in the object space at pass  $i$  could be a key for an embedded association of depth  $i+1$ . As the number of passes increases, the allowable depth for embedded associations increases.

#### 4.2.3 Method Three: Multiple Passes over the Associations

The next attempt makes use of this last observation, and allows fully embedded associations by making garbage collection passes of the above type until nothing new is saved in the association space. That is, the association space is repeatedly scanned for supported associations until those associations saved at pass  $i$  are the same as those that had been saved by pass  $i + 1$ :

```

proc assoc-garbage-collect3 () /* multiple scans of assoc space */
begin
  osgc-protect(root-node); /* mark all root-accessible objects */
  no-more-saved := false; /* ensure first pass */
  until no-more-saved = true do
    begin
      no-more-saved := true; /* default to all done */
      for each association, assoc, in associative memory do
        if not-marked-deleted?(assoc)
          and not-previously-amgc-saved?(assoc) /*!!! don't duplicate work: */
            then if assoc-supported?(assoc)
              then begin
                no-more-saved := false; /*!!! note work done */
                osgc-protect(assoc.value);
                amgc-save(assoc);
              end;
            end;
      end;
    end assoc-garbage-collect3.

```

The flag "no-more-saved" is used to test if any new associations have been saved during a pass of the main garbage collection loop. When this flag has the value true, no further associations have been saved

during the previous pass, and so the loop is exited; at this point, all accessible associations have been saved.

This third method performs all garbage collection desired, but it is horribly inefficient. In the worst possible scenario, for an associative memory containing  $\Lambda$  associations, it could require  $\Lambda$  passes over the all the association cells (a total of  $\Lambda^2$  cell examinations).

#### 4.2.4 Method Four: Dependency Threads for Unsupported Associations

The fourth option is the desired solution to the problem of garbage collecting embedded associations. One central garbage collection pass is made, and as previously unsupported associations become supported (due to protections in the object space of association key components) they are saved by the amgc. The newly supported associations (which had been previously unsupported) are found by means of *dependency threads*, unique data structures which are each maintained in a fragmented manner within those associations which are on the thread, as described in section 3.3.

The dependency thread points to all those associations which depended upon obj-1, for example. That is, as the amgc examined an association X which had obj-1 as one of its keys, obj-1 was tested to see if it had been accessible as of yet (protected by the osgc); if it had not yet been proven accessible, that association X was linked onto a dependency thread for object obj-1. When and if the object obj-1 gets reassigned and proven accessible, those associations on its dependency threads will be re-examined (by a call from the osgc to the procedure *update-thread*) and tested to see if they are supported. Those associations found to be fully supported will be saved.

```

proc assoc-garbage-collect4 () /* use dependency threads */
begin
  osgc-protect(root-node); /* mark all root-accessible objects */
  supported-assoc-list := empty; /* init sup list to empty */
  /* STAGE 1 : one pass over association cells to init threads: */
  for each association, assoc, in associative memory do
    if not-marked-deleted?(assoc)
      then if assoc-supported?(assoc)
        then merge(supported-assoc-list,assoc) /*!!! add to list */
        else thread-assoc(assoc); /*!!! make dep. thrd */
  /* STAGE 2: save all associations that are supported: */
  for each association, assoc, in supported-assoc-list do /* FIFO scan */
    begin
      osgc-protect(assoc.value);
      amgc-save(assoc);
    end;
  end assoc-garbage-collect4.

```

For each key object of an association, the association either is partially supported by that key (i.e., the



key object has been found to be accessible by some means), or depends upon that object (its subsequent accessibility will help support the association). Each key of an association is tested as to whether it helps support the association; if it fails the test, the association is placed on a dependency thread for that key object. If and when the object is proven to be accessible by the object space garbage collector, all associations on the object's dependency thread are updated accordingly. The space for the links in this dependency thread is found in the association cells themselves, hence requiring no extra storage.

```

proc thread-assoc(assoc)    /* put assoc on dep thrd for each unsupp. key */
begin
  for each keyi in assoc do
    if not-osgc-protected?(assoc.keyi)
      then begin
        osgc-mark-threaded(assoc.keyi); /* make thread for this key */
        put-on-dependency-thread(assoc,assoc.keyi);
      end;
    end thread-assoc.

```

As the object space garbage collector is protecting an object and proving its accessibility (often due to an *osgc-protect* call from the the associative memory garbage collector to the object name space garbage collector), the *osgc* must check if that object had been previously allocated a dependency thread. If it had, the *osgc* must call the following procedure to let the associative memory garbage collector know that a previously inaccessible key object is now accessible, and so that those associations which depended upon that key may now be supported fully. Perhaps an association is not fully supported, as of yet; in that case, the association will remain attached to some other key's dependency thread:

```

proc update-thread(object,newloc) /* called from obj name space collector*/
begin
  for each link in thread do
    begin
      assoc := head-of-assoc(link); /* get assoc this link part of */
      if assoc-supported?(assoc)
        then then merge(supported-assoc-list,assoc);
      end;
    end update-thread.

```

The special global data structure *supported-assoc-list* ensures that newly supported associations are not immediately saved in the association space. If they were, a newly-supported association would have its value object immediately protected in the object space, which could cause a dependency thread to be examined, which might cause *another* newly-completed association to be saved and *its* value to be saved by another *osgc-protect* call, and so on. By delaying installation of supported associations, no control stack (for unbounded recursive calls of the *amgc* to the *osgc* and back) is necessary.

The *merge* function above adds one new association object to the *supported-assoc-list*. If the new

item is added to the front of the list, the list acts as a stack, and is processed in a last-in, first-out manner as for a depth-first search. If the new association is added at the end, the list is used as a first-in, first-out, breadth-first search queue. The choice of whether to use the list as a stack or a queue is an implementational detail; both methods require neither extra storage space for the list, nor an auxiliary control stack.

### 4.3 Virtual Names and Two-Level Lookups

As stated above, it is assumed that the object name space garbage collector reassigns its object names. For fast access, our associative memory is based upon a hashing scheme which uses linked list buckets for collision resolution. Apparently,  $O(\Lambda)$  rehashes of the data in the associative memory will be necessary for the storage of  $\Lambda$  associations, as all the keys (object names) are changed by the osgc as it reassigns object names. However, a scheme which maps keys onto *virtual names*, and these onto the actual associations, saves most of the rehashes:

The two-level look-up scheme with virtual names is based upon the observation that our associative memory problem is similar to that of storing and accessing a large, sparse  $k$ -dimensional matrix of values. Most of the non-null values occur in just a small number of the "rows" (or "columns") of this  $k$ -dimensional matrix! (Please see Figure 4-1.)

Hashing the object keys (to place the associations in the correct bucket) is assumed to be the most costly part of the associative memory's garbage collection, so the use of this function, due to the osgc's reassignment of object names, is minimized. Assuming that the values occurred in basically  $r$  rows and columns of our value matrix, naive re-hashing due to garbage collection would cause  $O(r^k)$  hashes (approximately the number of associations). A two-level look-up which mapped each object to a unique virtual name, and then mapped these to the corresponding value, would entail only  $O(r*k)$  re-hashes due to garbage collection (approximately the number of row or column "keys" for association).

This solution of virtual names is seen to really just efficiently sidestep the problem: associations all still based upon objects with unique names, and this solution merely ensures that the unique names are unchanging, as well.

(The virtual name for an object could be made part of the object itself. This *vname* could be automatically installed as the zeroth component of any object when it initially created by a constructor invocation; no mutators would be allowed to change an object's virtual name component. This strategy would save the virtual name resolution mechanism the work of retrieving the vname for each key object,

		key2											
		A	B	C	H	I	J	K	L	7	15	121	73
key1	A												
	B	*			*			*	*			*	
	C												
	H												
	I	*			*			*	*			*	
	J	*			*			*	*			*	
	K												
	L												
	7												
	15												
	121	*			*			*	*			*	
	73	*			*			*	*			*	

\* = association exists

Figure 4-1: Association Space as a Sparse Matrix

as the *vname* would be immediately obtainable from the object. The benefit from this strategy would be the savings in processing time for the associative memory. The cost of this method would be that each object would have to have storage (somewhere) for a *vname* component; even those objects that were never used as keys for association must reserve space, since they could be used as association keys in the future. This space loss is potentially enormous, and so this method is not considered. Note that this is similar to the implementation of property lists in some classical LISP systems. In these systems, for each symbol, storage space was reserved for a pointer to the symbol's linear list of (property,value) pairs.)

This two-level look-up is straight-forward to implement directly from its definition. Note that at the association unification level of the *find* procedure, which locates the association, the bucket is found using the virtual names for the keys, but the actual *linear-search* through the bucket scans for a match on the originally specified object names. This ensures that the support for an association may be quickly found during garbage collection, even though a key object's name cannot be directly discerned from its virtual name.

```

proc find(key-list) returns association pointer assoc
begin
  /* STEP 1: find each virtual name (level 1) */
  for each keyi in key-list do
    begin
      bucket-vnr := hashtable-vnr[hash-vnr(keyi)];
      vnr-obj := linear-search(bucket-vnr,keyi);
      if vnr-obj = null
        then begin
          vnamei := uniquename(); /* install vname for new key : */
          insert-new-key-and-vname(keyi,vnamei);
        end;
      else vnamei := vnr-obj.vname;
    end;
  /* STEP 2: find association (level 2) */
  bucket-aun := hashtable-aun[hash-aun(vname-list)];
  assoc := linear-search(bucket2,key-list);
  return(assoc);
end find.

```

The hash tables *hashtab-vnr* and *hashtab-aun* (for virtual name resolution and association unification processing, respectively) are referenced as linear arrays. The functions *hash-vnr* and *hash-aun* hash their inputs to compute the indices within the appropriate hash tables. Within each bucket of the hash tables, a simple linear search is performed to find the desired cell:

```

proc linear-search(bucket,match-list) returns pointer ptrx
begin
  ptrx := null; /* default to not found */
  until empty(bucket)
  or not(ptrx = null) do
    begin
      next-elt := next element in bucket; /* try next one */
      if not-marked-deleted?(next-elt) /*make sure good elt */
      and for all i match-listi = next-elt.keyi
      then ptrx := next-elt;
    end;
  return(ptrx);
end linear-search.

```

This two-level strategy of virtual name resolution and association unification is independent of the dimension  $k$  of the associations. For our system, the  $k$  is two, but the method is efficient for all  $k$  greater than one.

None of the rehashing and vnr level look-up would have been necessary had the object name space garbage collector not modified the access paths of its objects. In that case, the object name itself would be the desired virtual name and would never change. As the *osgc*, previously incorporated into the system, does indeed use a compacting algorithm which modifies all the object names (which, in our case, are

virtual addresses), the added work of a two-level associative look-up, with re-hashing in the vnr level during the associative memory garbage collection, is necessary.

#### 4.4 LISP Databases

Our value of two for  $k$  was chosen to correspond with the notion of property lists in the language LISP, which is the basic mechanism from which LISP databases are developed. The use of the property list mechanism to support LISP databases is more fully explored in the paper by Sandewall [31].

The value of two for the dimension of our system is completely general, in the sense that functions *putk* and *getk*, for associations of  $k$  dimensions, may be coded easily in terms of multiple invocations of the 2-dimensional *put* and *get*. Unfortunately, *removek* cannot be defined unambiguously, for it is unclear whether only the final ( $k-1^{\text{st}}$ ) 2-dimensional association should be deleted, or whether all  $k-1$  associations which support the complete association should be removed.

(One assumption made was that searching for a non-existent association would produce the LISP constant *nil* as the default null value. This value *nil* may also be assigned as the property of an object, hence compounding the "null value problem" for databases [44]. That is, if an associative lookup returns the value *nil*, it may be interpreted as either: 1) the value for the association is unknown at this time, for it has yet to be *put* in the associative memory; 2) the value is unknowable, for all time; 3) the value is the logically false constant; 4) the value is the empty list; or 5) the value is the null symbol.)

## 5. Data Structures and Algorithms

This chapter presents the specific data structures and detailed algorithms for one particular implementation of the ideas put forth in the last chapter. The environment for the implementation is one in which object names are virtual addresses, and the objects are LISP consells or arrays (of various sizes). The associative memory supports a generalized property list facility.

All functions for the associative memory are presented in the Appendix. The language used is LISP, with a little bit of syntactic sugar. All the functions have been tested on a simulated associative memory device, and will be incorporated into the next iteration of the SCHEME system VLSI project [37, 18].

### 5.1 Data Structures for the Associative Memory

There are four basic data structures within the associative memory. Two of these are arrays for hash tables for each of the two levels of the lookup, and two of the data structures are "cells" which contain the appropriate association data and maintain the linked list structure for hashing collision resolution.

Each element of a hash table is a pointer to a bucket, which is the head of the linked list for hash collision resolution. All data stored in an array is accessed via these linked lists of cells. (Please see Figures 5-1 and 5-2 for the representation of the four data structures employed.)

The data structures which support processing at the virtual name resolution level are `hashtab-vnr` and `cell-vnr`. The hash table `hashtab-vnr` is referenced as an array, each of whose element points to the linked-list bucket. Each `cell-vnr` is a member of exactly one such bucket. Each such cell has components `nextcell`, a pointer to the next member of the same bucket, or a null pointer if the cell is the last in the bucket; `key`, the name of the object for which the cell contains the corresponding virtual name; and `vname`, that corresponding virtual name.

The data structures which support association unification level processing are similar. There is one hash table, `hashtab-aun`, which is an array of pointers to buckets. Each bucket contains a linked list of `cell-aun`'s, also called *association cells*. These include the components `nextcell`, as above; `key1` and `key2`, which are the names of the associated objects (the keys, with  $k = 2$ ); `hashnum`, the hash table index for the cell's bucket (to allow quick placement of the cell during garbage collection); and `val`, the value object of the association.

During garbage collection, the `cell-vnr`'s are modified to be the head of the dependency threads, as necessary. As such, a `cell-vnr` would contain a pointer to the thread as its `nextcell` component; the virtual

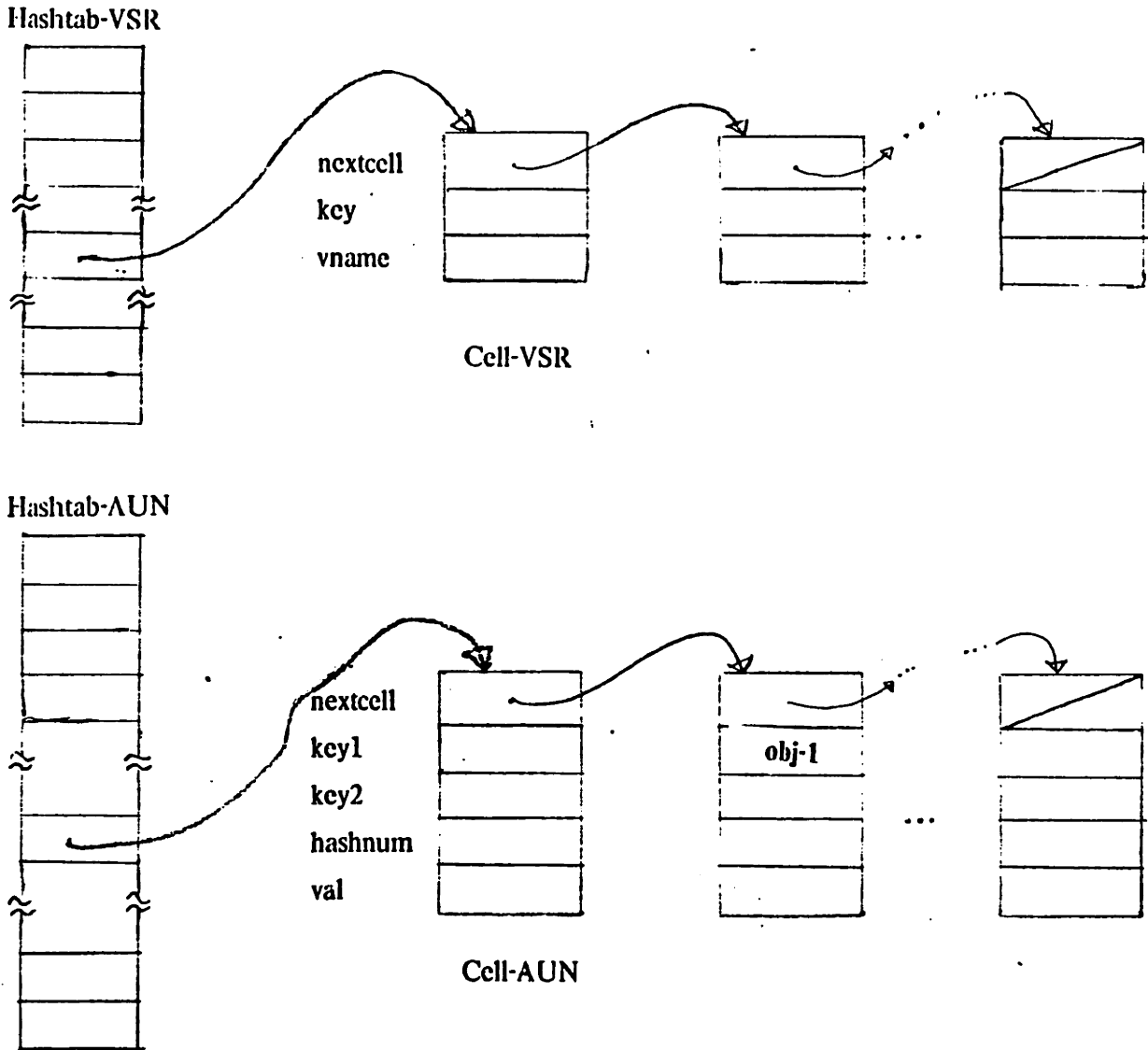


Figure 5-1: The Data Structures for the Associative Memory

name for the object (to ensure that it is not lost) in its usual **vname** spot; and a copy of the old first component of the object (in the cell's **key**), to give the object a place for a pointer to the thread.

In the garbage collection phase, the association cells are threaded in their **key** components, when necessary, on one or more dependency threads, and the **nextcell** component holds a count on the number of dependency threads with which an association is involved. Updated, newly supported associations which had been threaded are linked together via their **nextcell** components to form the list of completely supported associations.

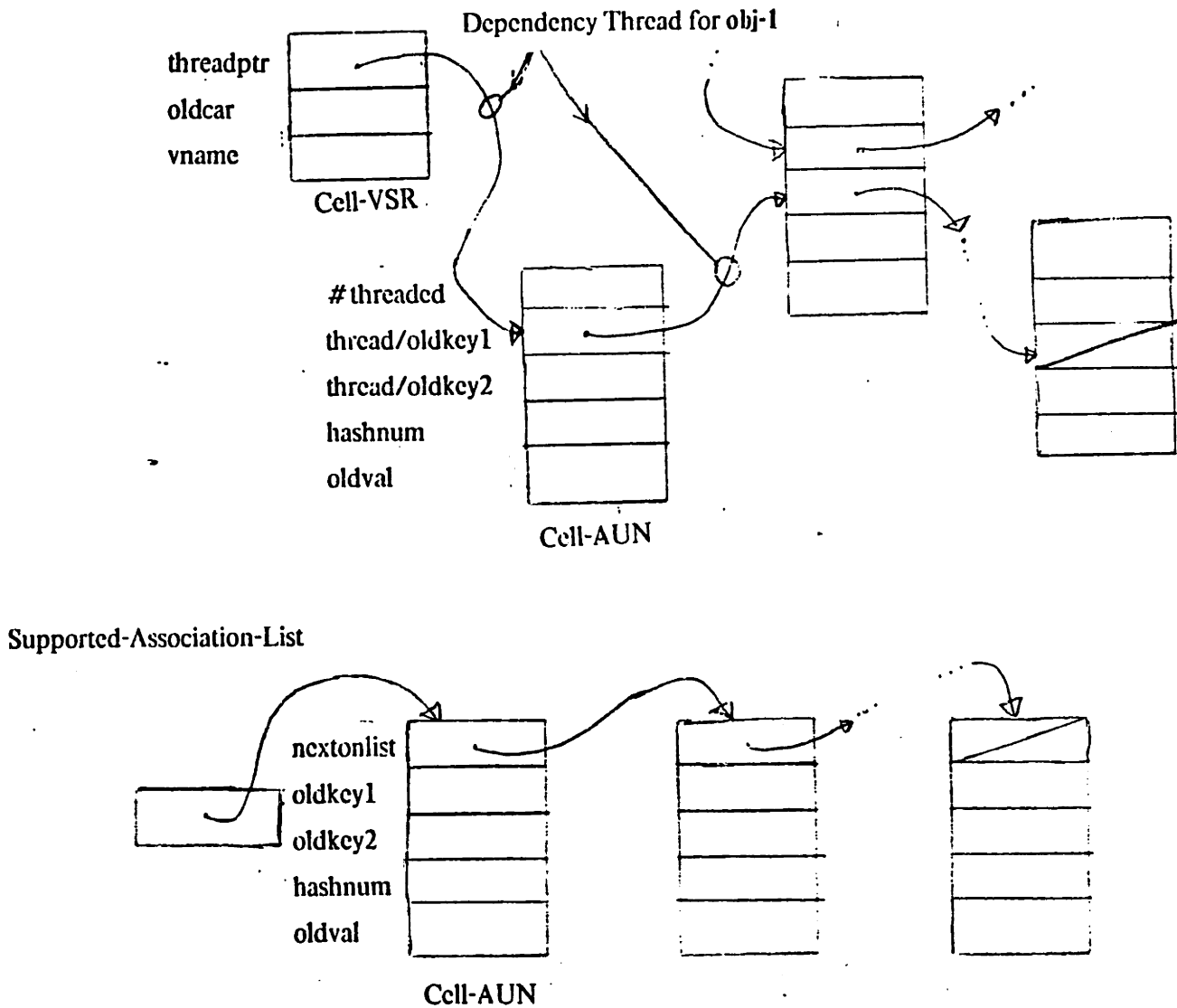


Figure 5-2: The Data Structures During Garbage Collection

A brief note on our terminology: a "chain" is regarded as a linked list structure, with each next element of the chain referenced by a pointer which is in a *fixed location* in the previous element. For example, the hashing resolution strategy employed uses chained buckets: each next element is referenced by the `nextcell` component of the previous element in the bucket, and this component is at a constant spot within the data structure (in our case, as the first component). The list of supported associations is similarly a chained structure, as the pointers which support the list are in the `nextcell` component of each resident association data structure on the list.

A "thread" is seen as a linking structure in which the pointer may reside at any one of a number of



locations within the data structure. This is seen in the dependency threads, where each thread weaves its way through unsupported associations.

We have found that our method of dependency threads is similar to some of the methods independently investigated by Morris [27] and Thorelli [39] for simple garbage collection. Their data structures were forward pointers used in compaction, and were not used at all regarding questions of associative memories or hash table garbage collection. Goto [16] has suggested a method of garbage-collecting an associative memory in which embedded associations are not allowed.

## 5.2 Algorithms

### 5.2.1 Associative Search Functions

The procedures *put*, *get* and *rem* all initially call the associative lookup procedure *find*. *Find* first searches for the virtual name associated with each object; if no virtual name yet exists for an object, one is created. *Find* then searches for the association cell corresponding to the given objects, by hashing the virtual names to get the aun level hash table's bucket, and scanning this bucket by matching against the original keys. The value found for the association, if any, is returned, and condition flags are set appropriately. *Put* updates and returns the (previous) found value, or instantiates a new association cell if no value had been found; *get* returns the value found; *rem* deletes the association and returns the old value: (The procedure *find* is as above.)

```

proc put(key1-obj, key2-obj, val-obj) returns pointer oldval
begin
  assoc := find(key1-obj, key2-obj);
  if assoc = null
  then begin /* make new association */
    make-new-assoc(key1-obj, key2-obj, val-obj);
    oldval := null;
  end
  else begin /* modify old association */
    oldval := assoc.value; /* return old value (arbitrary convention) */
    assoc.value := val-obj;
  end;
  return oldval;
end put.

```

```

proc get(key1-obj,key2-obj) returns pointer oldval
begin
  assoc := find(key1-obj,key2-obj);
  if assoc = null
    then oldval := null
    else oldval := assoc.value;
  return oldval;
end get.

```

```

proc rem(key1-obj,key2-obj) returns pointer oldval
begin
  assoc := find(key1-obj,key2-obj);
  if assoc = null
    then oldval := null
    else begin
      oldval := assoc.value;
      mark-deleted(assoc);
    end;
  return oldval;
end rem.

```

### 5.2.2 Garbage Collection Processing

The object name space garbage collector must be simply modified to interact properly with the associative memory garbage collector. The osgc must be able to recognize objects that have been assigned dependency threads, and must call the *amgc* to invoke the *update-thread* function. A few lines of code must be added to *copy-old-to-new*.

```

proc copy-old-to-new(oldptr) returns pointer newptr
begin
  oldobj := object-in-obspace(oldptr);
  if already-reassigned?(oldobj)
    then newptr := new-assignment(oldobj) /* point to copy in newspace */
    else begin
      /* or make a new copy */
      if threaded?(oldobj) /*!!! if threaded */
        then update-thread(oldobj,freeptr); /* !!! then update thread*/
      for each component, comp, in oldobj do
        copy-component(oldobj,comp,freeptr); /*copy all obj into newspace*/
        note-reassigned(oldobj,freeptr); /* tell new assignment */
        newptr := freeptr; /* new object assignment */
        freeptr := next-free-obj-name(freeptr); /* update freeptr */
      end;
    return(newptr);
  end copy-old-to-new.

```

The garbage collecting algorithm for the associative memory copies supported associations from the old association space to the new association space, and proceeds in stages. In the initialization section, space for a new *hashtab-vnr* and *hashtab-aun* is created. (The following code gives a bit more detail than previously; see Appendix for complete definitions of functions in LISP.)

```

proc assoc-garbage-collect() /* full amgc */
begin
/*INITIALIZATION : */
create-new-hashtabs(); /* make space for new hash tables */
osgc-protect(root-node); /* mark all root-accessible objects */
supported-assoc-list := empty; /* init sup list to empty */
/* STAGE 0 : one scan of the old key-vname structures */
for each hash bucket, bucket1, in hashtab-vnr do
for each cell-vnr, cell1, in bucket1 do
if already-reassigned?(cell1)
then insert-new-key-and-vname(new-assignment(cell1.key),cell1.vname)
else create-dep-thread-head(cell1);
/* STAGE 1 : one pass over association cells to init threads: */
for each hash bucket, bucket2, in hashtab-aun do
for each association cell-aun, cell2, in bucket2 do
if not-marked-deleted?(cell2)
then if assoc-supported?(cell2)
then merge(supported-assoc-list,cell2) /*!!! add to list */
else thread-assoc(cell2); /*!!! make dep. thrd */
/* STAGE 2: save all associations that are supported: */
for each association, cell2, in supported-assoc-list do /* FIFO scan */
begin
osgc-protect(cell2.value);
amgc-save(cell2);
end;
end assoc-garbage-collect.

```

In Stage0, the mappings from object names to virtual names which are stored in the old `hashtab-vnr` are searched in an iterative manner, running down the linked list for each hash bucket. Object names which had been reassigned by the `osgc`, and are therefore accessible in the future, are re-hashed and stored in the new `hashtab-vnr`. This re-hashing is necessary because the object names had been reassigned, of course.

Object names which had not been reassigned have empty dependency threads created for them. The `cell-vnr`, which had previously stored the mapping from object to virtual name, now becomes the header for the dependency thread, and stores the datum which had resided in the first component of the old object. That object is now free to point to the dependency chain header. (We could have used a hashing technique on object names themselves to retrieve thread heads, but installing a pointer to the thread head as the object's first component is more efficient.) The `nextcell` pointer of the header `cell-vnr` will point to all those associations on the thread (there are none as of yet). Associations are put on a dependency chain for an object when it is seen that the object, as one key of the association, has not been reassigned, and hence that association is dependent upon that object.

The next stage, Stage1 of the garbage collector, scans through the old array `hashtab-aun`. It

iteratively searches through each bucket looking for each association which is completely supported (all key objects in the association had been reassigned). Each such association will be eventually copied into the association space, and is immediately put on the list for completely supported associations. (Rehashing is not necessary as the identifiers for the buckets of the hashtable-aun are the virtual names for the objects, which are unique and constant.) Incomplete associations are linked onto a dependency thread for each object in the association which had not been reassigned in the object name space.

Stage2 of the garbage collector scans through all the cells in the supported association list. As each such association in the list had previously been found to be fully supported, it is saved by the amgc; as an association cell is saved, a call goes to the object space garbage collector to ensure that the value for the association is protected (and reassigned). If this value object had been previously allocated a dependency thread, the object space garbage collector will request an unthreading for this object.

When the unthreading of a dependency chain is requested by the osgc, the value for the first component in the object (the car of the LISP conscell, which we had stolen) is returned to the osgc by the amgc. The dependency thread is searched, and all fully supported associations found at this time are placed on the supported-assoc-list. (As an implementational detail, the old key is placed in each association on the dependency chain, returning the original contents of each.) No extra space for a control stack or association storage is required, as the list itself is stored in the association cells as remnants of previous dependency threads, and the use of the list prevents unbounded recursive calls between the osgc and the amgc.

In all, the following has been demonstrated:

*Theorem.* The garbage collection and search algorithms, presented in the Appendix, work correctly: All accessible associations are saved by the garbage collector, all inaccessible associations are destroyed, and associations are properly found, updated, and deleted upon demand. The garbage collector works in time  $O(A+H)$ , where  $H$  is the size of the hash table hashtable-aun, and  $A$  is the number of associations stored at the time of garbage collection.

*Proof.* As shown above, the search algorithms simply do a standard hash lookup for two levels. The garbage collector copies completely supported associations (places onto the supported-assoc-list) and threads unsupported associations. As these latter become completely supported, they are placed on the list of supported associations, which is eventually moved into the new association space.

Each association structure cell-aun is examined at most  $k + 2$  times: initially, perhaps once to

unthread for each key, and once to put in the new association space. The number of first level virtual names is assumed to be on the order of the  $k^{\text{th}}$  root of  $\Lambda$ , and each cell-vnr is processed at most three times: initially as the dependency threads are created, once for updating each dependency thread, and once for the final installation. (The thread pointer for each dependency chain is modified as new associations are threaded; this cost was counted as part of the cell-aun processing.) The first level hash table, `hashtab-vnr`, is smaller than `hashtab-aun`, and each hash table is examined linearly, scanning through each bucket.■

## 6. Associations as Objects

Associations are considered as full-fledged objects; association-id's will reside in the object name space. Ramifications of this change are discussed, and modifications to the garbage collection algorithms are presented.

### 6.1 Object Functions

As stated above, objects are created by means of constructors and modified by mutators. Selectors retrieve the internal components of objects.

Objects may not be explicitly deleted. An object goes away if it is not accessible; the garbage collector implicitly removes objects from the object space as necessary.

### 6.2 Association Operations

We shall now consider general associations as full-fledged objects which reside in the object space, removing the assumption in effect in previous chapters. We also now assume that an association may be formed over any integral number of objects as keys, though still yielding only one object as a value; the notion of the constant dimension  $k$  of an associative system will no longer be considered.

To create an association object, a constructor function *make-assoc* is appropriate. This function must return the object name of the created association object, so that the association may be referenced as an object. *Make-assoc* must return a *unique instance* of the association. That is, there cannot be two instances of an association over objects A and B (in that order). Therefore, *make-assoc* forms a new association only if the key objects had not been associated before (else, an error occurs). It is seen that *make-assoc* works as our previous function *put* if the association object had not previously existed.

As each association is unique, if two association objects have the same  $key_i$  components, they must be the same object. That is, for association objects A and B, if A is equivalent to B, then A is identical to B.

This is similar to the unique *cons* function *ucons* found in the LISP-like language BRANDX [38, 16]. The BRANDX call *ucons(a,b)* returns a unique conscell whose car is a, and whose cdr is b. Another identical call will return the *same* conscell, not just a copy as in regular LISP.

The selectors appropriate to association objects are *number-of-keys*, *assoc-value*, *first-key*, *second-key*, et cetera. As usual, these selectors must be given pointers to the association objects, and make no use of the association mapping mechanism. There may be some implementation-dependent system information

within the association objects (for example, the aun level hash table index) which is unavailable to the user by means of the given selectors.

The *number-of-keys* and *key<sub>i</sub>* components in an association object may not be changed during the existence of the association. The association value component is modified by means of the mutator *modify-assoc*. Note that this mutator does not reference objects by direct pointers to them, but rather uses indirect pointers (the keys) via the association mapping mechanism to find the association object of interest.

The function *get-assoc* must be included in our package; this function returns the association object of interest, if it exists, extracted through the use of the association mapping mechanism. It is like the old function *get*, except that *get* just returned the value of the association, whereas the procedure *get-assoc* returns the entire object. (This procedure is not a selector as it does not return a component, but rather a pointer to the entire association object.) The composite selector *get-value* is to be included for convenience (and similarity to the old *get*), and is defined to be an *assoc-value* selector applied to the object retrieved via *get-assoc*.

There is no function similar to the old *remove*, for the explicit deletion of objects is not allowed. The mutator *reset-assoc* is desired; this function would reset the value component of the association object to the null object. Therefore, use of the function *get-assoc* after the function *reset-assoc* would return the null object as a value, exactly as the old function *remove* used to perform.

### 6.3 Sequencing

The association mapping mechanism finds the name of the unique association object corresponding to a given sequence of constants and object names (keys). If  $D$  is the space of object names unioned with the space of constants, and  $\mathcal{A}$  is the space of association-id's, a subset of the object name space, then the association functions uniquely determine the mapping of  $D^*$  onto  $\mathcal{A}$ .

This mapping of a list of objects (treating constants as objects for the moment) to a unique object (association) is known as *sequencing*. This is analogous to the use of generalized pairing functions on the integers.

### 6.4 Generalized Pairing Functions

A *pairing function* is a unique, reversible mapping of a pair of natural numbers into one natural number [23]. This mapping loses no information, as may be seen: if  $f$  is a pairing function with inverses

$f_1$  and  $f_2$ , for natural numbers  $x$  and  $y$ , if  $f(x,y) = z$ , then  $f_1(z) = x$ , and  $f_2(z) = y$ . The function  $f$  is used to encode the numbers  $x$  and  $y$  as a single integer,  $z$ , such that the original integers may be easily retrieved.

This encoding is analogous to the way that the association mapping mechanism will map a sequence of object names into a unique object name, such that the original keys are easily recovered by use of the selectors.

A generalized pairing function maps an arbitrarily long ordered list of integers into a single integer, such that no information is lost, as above. A simple example of a function of this type, with integral inputs  $x_i$ , would be the function that first finds the product of the terms  $p_i^{x_i}$  (with  $p_i$  the  $i^{\text{th}}$  prime number), and maps this single rational number onto the positive integers by the standard diagonalization methods. The original numbers could be retrieved from this resultant integer by reversing the diagonalization and then calculating discrete logarithms,

$$\#_b(n) = \max \{i \mid b^i \mid n\}, \text{ with } x \mid y = \text{def } x \text{ evenly divides } y.$$

The term *sequencing* is due to the application of this function to encode sequences of Boolean variables (all of which similarly quantified by means of the existential or universal quantifier) into one Boolean variable (with the same quantifier as in the sequence) for use in quantifier-free Boolean formulae.

## 6.5 Garbage Collection of Association Objects

The philosophy of garbage collection remains as in the previous system: all accessible objects are to be saved, and inaccessible ones deleted. The algorithms presented in the previous section need be modified only a little to achieve this goal.

The first stage of the full garbage collector may perform a full MFY-style object name space garbage collection of all root-accessible object names. This would save all association objects that are root-accessible as well as the usual (non-association) root-accessible objects. The key objects corresponding to these root-accessible association objects would be saved, to ensure that future invocations of selector functions (like *second-key*) could not get object names for (references to) deleted objects.

The second stage would perform the vnr level scan (copying virtual names or creating dependency thread heads) on the private memory of the association mapping mechanism. These data structures are part of the system support mechanism, and do not appear in the object name space.

The third and final stage of the garbage collector scans through all the associations; if an association



object has yet to be saved (it is not root-accessible as an object), then it is saved if its keys are root-accessible objects, or else it is strung on various dependency threads. An association is saved in the object name space by an *osgc-protect(assoc)* invocation; the value of the association will be protected when the association object is seen by the *scanptr* in the new object name space, which may cause a dependency thread (for the object corresponding to the association's value component) to be examined.

It is important to note that as a dependency thread for an object is updated, newly supported associations are copied *immediately* into the object space using the MFY procedure *osgc-protect*. The new space now doubles as *both* the search queue for pointer-accessible objects and as a queue (a first-in, first-out implementation) for supported association objects with accessible keys. Therefore, a distinct **supported-assoc-list** is no longer necessary, for it is implicitly maintained in the new object name space. The value components for associations saved in this manner are not examined immediately, but rather are later examined as the *scanptr* sweeps through the space finding new, unresolved references. Again, no auxiliary control stack is needed by this procedure.

All root-accessible objects are saved in the first stage of this garbage collector. All association objects for associations of depth one are immediately saved in stage three, as are all deeper association objects as they are proven to be accessible. The final stage is seen to save just those objects which are accessible but not root-accessible (that is, those objects which are pointer-accessible from some association object of depth one or greater).

The depth of an association for an association object which is root-accessible may be defined to be zero. This leads to a consistent definition of "association depth" for all objects.

(The above algorithm is seen to be a solution to the single-source reachability problem for an AND-OR directed graph: given a root vertex, a list of edges, and a set of monotonic Boolean formulae which define the reachability of the vertices from their predecessors, which vertices may be reached from the root vertex?)

## 7. Future Work and Summary

### 7.1 Future Work

The data structures in the associative memory have been designed with the intent of making the associative memory sensitive to dynamic changes in the operating environment; for example, paging characteristics may dynamically determine the sizes of our hash tables (hashtab-vnr and hashtab-aun). In the future, we hope to make the associative memory extensible [12] to allow for compatibility with dynamic system resource utilization.

Further work needs to be done on the object model of computation. Its inherent limitations and its usefulness as a theoretical tool need to be explored. The object model may be seen as similar to both the Storage Modification Machine and the RAM models [33], and this similarity may be more closely examined.

To save space in the given implementation of the LISP property list facility, one could pack all the cell-aun's in one bucket into a contiguous block of memory. (This is similar to "CDR-coding" methods in LISP systems [36]; please see Figure 7-1 for an example of this blocking.) This would save the real resource used by each cell-aun's nextcell pointer, which frequently just points to the next physical cell in the association space.

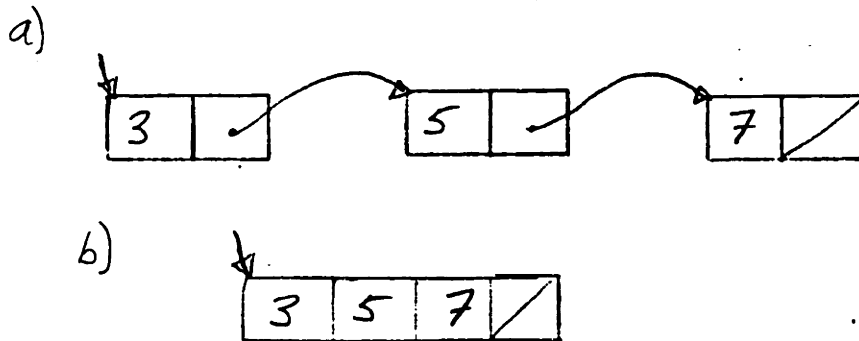


Figure 7-1: CDR-Coding

a) before and b) after CDR-coding

### 7.2 Summary

Efficient algorithms for an associative memory in a garbage collecting environment have been demonstrated. Our method is useful for many types of abstract objects which may be garbage collected,

and for which speedy access to associations is desired. This system was designed for, but is not limited to, compatibility with LISP databases implemented via the property list mechanism.

## Appendix

Following is an implementation of a garbage-collecting associative memory to support the LISP property list facility. The memory space is simulated as a linear array, and the simulation itself is built upon a working LISP system. The data structures used have been depicted in Figures 5-1 and 5-2; the algorithms given use the dependency thread and list of supported associations techniques as described in the previous text.



A Garbage-Collecting Associative Memory for LISP  
(to support the property list facility)

Richard A. Ross

(rickr@ai)  
MIT NE43-836  
(617) 253-7843

This file contains code for a garbage-collecting associative memory. Memory is modelled as a linear array. A two-level associative lookup is employed. Each key is mapped to a unique virtual name, and then the set of virtual names is mapped to the desired value. Symbols are not interned.

This code runs on Lisp Machines and MACLISP systems.

the spaces (virtual names):

memspace - linear memory for conscells  
amspace - linear space for associations  
oldspace - previous memspace (for gc)  
oldamspace - previous amspace (for gc)  
(m1space, a1space, m2space, a2space are real names)

the registers (global vars):

root - topmost environment ptr in memspace  
free - freespace ptr  
amfree - amspace free ptr  
scanptr - for mfy gc  
freeptr - for mfy gc  
sal - supported assoc. list ptr  
freecnt - # free words left in memspace  
amsfreecnt - # free words left in amspace  
unique-name\*-cnt - new virtual name for assoc key

system constants:

xnil - nil = 0  
size-cell1 - size of vnr cell for assoc  
size-cell2 - size of aun cell for assoc  
size-array1 - size of vnr hash table  
size-array2 - size of aun hash table  
base-array1 - base of vnr hashtable in amspace  
base-array2 - base of aun hashtable in amspace

word structure:

words in memspace:

tdatum - datum stored in word  
tatom - = T iff word is first part of atom (symbol)  
tlist - = T iff word is first part of list  
trelocate - = T iff gc relocated (& saved) conscell  
tthread - = T iff word is thread ptr (in gc)

words in amspace:

tobj - datum stored in word  
tnum - relative displacement of word in assoc  
tcell - = T iff word is part of a cell (& not array)  
ttype2 - = T iff word is part of assoc (aun cell)  
tdelete - = T iff word is first part of rem'ed assoc

```

:
: initialization fcns
:
(defun initspace (n)
  (defspaces)
  (defarray m1space n (numtypes memspace))
  (defarray m2space n (numtypes memspace))
  (defarray a1space n (numtypes amspace))
  (defarray a2space n (numtypes amspace))
  (putprop 'memspace n 'numword)
  (putprop 'oldspace n 'numword)
  (putprop 'oldamspace n 'numword)
  (putprop 'amspace n 'numword)
  (setq memspace m1space)
  (setq oldspace m2space)
  (setq oldamspace a2space)
  (setq amspace a1space)
  (unique-name* 'init)
  (initxn1l)
  (initamspace))

(defun initamspace ()
  (setq size-cell2 5)
  (setq size-cell1 3)
  (setq base-array1 1)
  (setq size-array1 (// (numwords amspace) 6))
  (setq base-array2 (+ base-array1 size-array1))
  (clrarray amspace)
  (setq size-array2 (// (numwords amspace) 4))
  (setq amfree (+ base-array2 size-array2))
  (setq amfreecnt (- (numwords amspace) amfree)))

(defun initxn1l ()
  (setq free 0)
  (setq freecnt (numwords memspace))
  (xconsx 0 0) ;set up xn1l
  (setq root xn1l)
  (settype memspace xn1l tatom)
  (settype memspace xn1l tlist))

(defun initroot ()(initxn1l)) ;alias

(defun defspaces() ;define spaces as array of tagged words
  (defspace memspace tdatum tatom tlist trelocate tthread)
  (defspace oldspace tdatum tatom tlist trelocate tthread)
  (defspace amspace tobj tnum tcell ttype2 tdelete)
  (defspace oldamspace tobj tnum tcell ttype2 tdelete))

```

```

:
; assoc mem fcns with full two-level lookup
:

(defun xget (k1 k2)
  (let ((c (amfind k1 k2))) ;c = cons of cell&hashnum
    (cond ((cellfound? c) (xval (car c)))
          (t xnil))))

(defun xput (k1 k2 v)
  (let ((c (amfind k1 k2))) ;c = cons of cell&hashnum
    (cond ((cellfound? c) (prog1 (xval (car c)) (setval v (car c))))
          (t (xmake-assoc k1 k2 v (cdr c) xnil))))))

(defun xrem (k1 k2)
  (let ((c (amfind k1 k2))) ;c = cons of cell&hashnum
    (cond ((cellfound? c) (mark-delete (car c)) (xval (car c)))
          (t xnil))))

(defun cellfound?(x) (not (xnull (car x))))

(defun amfind (k1 k2) ;send back cell or place for new one
  (setq hashnum (hash2 (find-or-make-vname k1)
                       (find-or-make-vname k2)))
  (do ((bucket (arref amspace (+ base-array2 hashnum) tobj) (xnext bucket)))
      ((cond ((xnull bucket) (setq bucket xnil))
            ((and (not (deleted? bucket))
                  (eq k1 (xkey1 bucket))
                  (eq k2 (xkey2 bucket))) bucket))
    (cons bucket hashnum))
  ()))

(defun find-or-make-vname (k)
  (let ((hashn1 (hash1 k)))
    (let ((c1
           (do ((bucket1 (arref amspace (+ base-array1 hashn1) tobj) (xnext bucket1)))
               ((or (xnull bucket1) (eq k (xkey bucket1))) bucket1)
              ())))
      (cond ((xnull c1) (make-new-vname k hashn1))
            (t (xvname c1))))))

(defun xmake-assoc (k1 k2 v hashn)
  (let ((n amfree))
    (incr amfree size-cell2)
    (decr amfreecnt size-cell2)
    (arset amspace k1 (1+ n) tobj)
    (arset amspace k2 (+ 2 n) tobj)
    (arset amspace hashn (+ 3 n) tobj)
    (arset amspace v (+ 4 n) tobj)
    (settype amspace n tcell)
    (settype amspace n ttype2)
    (setnums2 n)
    (link base-array2 hashn n)))

(defun make-new-vname (kx hx)
  (let ((vnamex (unique-name*)))
    (insert-vname kx hx vnamex)
    vnamex))

(defun insert-vname (kx hx vnamex)
  (arset amspace kx (1+ amfree) tobj)
  (arset amspace vnamex (+ 2 amfree) tobj)
  (settype amspace amfree tcell)
  (clrtype amspace amfree ttype2) ;we're type 1
  (link base-array1 hx amfree)
  (incr amfree size-cell1)
  (decr amfreecnt size-cell1)).

```



```

(defun link (arraybase hnum cel)
  (arset amspace (arref amspace (+ arraybase hnum) tobj) cel tobj) :set next
  (arset amspace cel (+ arraybase hnum) tobj)) ;link us up

(defun setnums2 (cel)
  (do ((i 0 (1+ i)))
      ((= i size-cel12) ())
    (arset amspace i (+ i cel) tnum))) ;set num fields in assocs

(defun hash1 (n) (\ (lsh n -2) size-array1))
(defun hash2 (a b) (\ (logxor a b) size-array2))

(defun unique-name* (&optional (flag nil))
  (cond ((eq flag 'init) (setq unique-name*-cnt 0))
        (t (incr unique-name*-cnt))))

(defun xnext (n) (arref amspace n tobj))
(defun xkey1 (n) (arref amspace (1+ n) tobj))
(defun xkey2 (n) (arref amspace (+ 2 n) tobj))
(defun xhash (n) (arref amspace (+ 3 n) tobj))
(defun xval (n) (arref amspace (+ 4 n) tobj))
(defun setval (n x) (arset amspace x (+ 4 n) tobj))

(defun mark-delete (n) (settype amspace n tdelete))
(defun deleted? (n) (typeset? amspace n tdelete))

(defun xkey (n) (arref amspace (1+ n) tobj))
(defun xvname (n) (arref amspace (+ 2 n) tobj))

;
; def's for oldamspace
;

(defun oldnext (n) (arref oldamspace n tobj))
(defun oldkey1 (n) (arref oldamspace (1+ n) tobj))
(defun oldkey2 (n) (arref oldamspace (+ 2 n) tobj))
(defun oldhash (n) (arref oldamspace (+ 3 n) tobj))
(defun oldval (n) (arref oldamspace (+ 4 n) tobj))

(defun oldkey (n) (arref oldamspace (1+ n) tobj))
(defun oldvname (n) (arref oldamspace (+ 2 n) tobj))

```

```

:
:
:   the amspace garbage collector
:
:
(defun amgc ()
  (flip-spaces)
  (osgc-protect root)
  (setq sal xnil)                                ;supported assoc. list empty
  (stage0)
  (stage1)
  (stage2)
  (setq freecnt (- (numwords memspace) (setq free freeptr)))
  (setq amfreecnt (- (numwords amspace) amfree)))

(defun stage0 ()
  (let (b1 c1 nextc1)
    (setq b1 0)
    (untildo (= b1 size-array1)
      (progn
        (setq c1 (arref oldamspace (+ base-array1 b1) tobj))
        (untildo (xnull c1)
          (progn
            (setq nextc1 (oldnext c1))
            (cond ((relocated? (oldkey c1))
                  (let ((n1 (new-location (oldkey c1))))
                    (insert-vname n1
                                   (hash1 n1)
                                   (oldvname c1))))
              (t (make-dep-thread-head c1)))
            (setq c1 nextc1)))
        (incr b1))))))

(defun stage1 ()
  (let (b2 c2 nextc2)
    (setq b2 0)
    (untildo (= b2 size-array2)
      (progn
        (setq c2 (arref oldamspace (+ base-array2 b2) tobj))
        (untildo (xnull c2)
          (progn
            (setq nextc2 (oldnext c2))
            (cond ((old-deleted? c2) ())
                  ((assoc-supported? c2)
                   (merge-into-sal c2))
                  (t (thread-it c2)))
            (setq c2 nextc2)))
        (incr b2))))))

(defun stage2 ()
  (let (oldassoc)
    (untildo (xnull sal)
      (progn
        (setq oldassoc sal)
        (setq sal (oldnext sal))                ;to allow for depth first w/ unthread
        (save (oldval oldassoc))
        (xmake-assoc (new-location (oldkey1 oldassoc))
                     (new-location (oldkey2 oldassoc))
                     (new-location (oldval oldassoc))
                     (oldhash oldassoc))))))    ;hash num over virtual names

```

```

(defun make-dep-thread-head (c1x)
  (let ((k1 (oldkey c1x)))
    (arset oldamspace (oldcarx k1) (1+ c1x) tobj) ;steal the car
    (arset oldspace c1x k1 tdatum) ;make it point to us
    (settype oldspace k1 tthread) ;mark that it's threaded
    (set-oldthread c1x xnil))) ;no assoc's in thread

(defun thread-head (key) (arref oldspace key tdatum))

(defun return-car (c1x place)
  (arset oldspace (arref oldamspace (1+ c1x) tobj) place tdatum)
  (clrtype oldspace place tthread))

(defun unthread (oldloc newloc)
  (let (cell1 threadlink nextlink cell2)
    (setq cell1 (thread-head oldloc))
    (setq threadlink (oldthread cell1))
    (unfildo (xnull threadlink)
      (progn
        (setq nextlink (arref oldamspace threadlink tobj)) ;next on thread
        (arset oldamspace oldloc threadlink tobj) ;reset old key
        (setq cell2 (- threadlink (arref oldamspace threadlink tnum)))
        (decr-oldcount cell2)
        (cond ((zero-oldcount? cell2) (merge-into-sal cell2)))
        (setq threadlink nextlink)))
      (return-car cell1 oldloc)
      (insert-vname newloc (hash1 newloc) (oldvname cell1))))

(defun assoc-supported? (assoc)
  (and (relocated? (oldkey1 assoc))
       (relocated? (oldkey2 assoc))))

(defun merge-into-sal (assoc) ;sal as stack
  (arset oldamspace sal assoc tobj) ;our next is old top
  (setq sal assoc)) ;we're new top

(defun thread-it (assoc)
  (set-oldcount assoc 0)
  (cond ((not (relocated? (oldkey1 assoc)))
        (enthread (1+ assoc) (oldkey1 assoc))
        (incr-oldcount assoc)))
  (cond ((not (relocated? (oldkey2 assoc)))
        (enthread (+ 2 assoc) (oldkey2 assoc))
        (incr-oldcount assoc))))

(defun enthread (place key)
  (let ((c1 (thread-head key)))
    (arset oldamspace (oldthread c1) place tobj)
    (set-oldthread c1 place)))

(defun set-oldcount (n x) (arset oldamspace x n tobj))
(defun incr-oldcount (n) (arset oldamspace (1+ (arref oldamspace n tobj)) n tobj))
(defun decr-oldcount (n) (arset oldamspace (1- (arref oldamspace n tobj)) n tobj))
(defun zero-oldcount? (n) (zerop (arref oldamspace n tobj)))

(defun old-deleted? (n) (typeset? oldamspace n tdelete))

(defun oldthread (n) (arref oldamspace n tobj))
(defun set-oldthread (n x) (arset oldamspace x n tobj))

```

```

:
: the myf garbage collector
:

(defun gc ()
  (flip-spaces)
  (setq root (save root))
  (setq freecnt (- (numwords memspace) (setq free freeptr))))
;just gc memspace, don't save assocs

(defun flip-spaces ()
  (cond ((eq memspace m1space)
    (setq oldspace m1space)
    (setq memspace m2space)
    (setq oldamspace a1space)
    (setq amspace a2space))
    ((eq memspace m2space)
    (setq oldspace m2space)
    (setq memspace m1space)
    (setq oldamspace a2space)
    (setq amspace a1space)))
  (copy-xnil) ;save our nil (collect valuable prizes)
  (setq scanptr (setq freeptr 2))
  (initamspace))

(defun copy-xnil ()
  (initxnil)
  (set-new-loc xnil xnil)
  (setq scanptr (setq freeptr 2)))

(defun save (rootptr)
  (progn (copyptr rootptr)
    (untildo (>= scanptr freeptr)
      (cond ((xatomp scanptr) (incr scanptr 2))
            (t (setdatum scanptr (copyptr (xdatum scanptr))
              (incr scanptr)))))))

(defmacro osgc-protect (ptr) `(setq ,ptr (save ,ptr)))

(defun relocated? (ptr) (typeset? oldspace ptr trelocate))
(defun new-location (ptr) (oldcarx ptr))
(defun set-new-loc (ptr newloc)
  (settype oldspace ptr trelocate)
  (arset oldspace newloc ptr tdatum))

(defun copyptr (ptr)
  (cond ((relocated? ptr) (new-location ptr))
    ((typeset? oldspace ptr tthread) (unthread ptr freeptr) (copyptr ptr))
    (t (progn freeptr
      (copyword oldspace ptr memspace freeptr)
      (set-new-loc ptr freeptr) ;note where it went
      (incr freeptr)
      (copyword oldspace (1+ ptr) memspace freeptr)
      (incr freeptr)))))

(defun oldcarx (n) (arref oldspace n tdatum))
(defun oldcdrx (n) (arref oldspace (1+ n) tdatum))

```

```

:
:   standard LISP memory fcns
:
:                                     (note symbols not interned; atoms are objects)

(defun xconsx (a b)
  (progl (setq temp free)
         (incr free 2)                               ;note space used
         (decr freecnt 2)
         (settype memspace temp tlist)
         (clrtype memspace temp tatom)
         (clrtype memspace temp tthread)
         (clrtype memspace temp trelocate)
         (xrplaca temp a)
         (xrplacd temp b)))

(defun xcar (n) (cond ((xnull n) n)
                     ((xlistp n) (arref memspace n tdatum))
                     (t (error "in xcar -- not xlist" n))))

(defun xcdr (n) (cond ((xnull n) n)
                     ((xlistp n) (arref memspace (1+ n) tdatum))
                     (t (error "in xcdr -- not xlist" n))))

(defun xrplaca (n a)
  (cond ((not (xobj? a)) (error "in xrplaca, not xobj" a))
        ((not (xlistp n)) (error "in xrplaca, not xlist" n))
        (t (arset memspace a n tdatum))))

(defun xrplacd (n b)
  (cond ((not (xobj? b)) (error "in xrplacd, not xobj" b))
        ((not (xlistp n)) (error "in xrplacd, not xlist" n))
        (t (arset memspace b (1+ n) tdatum))))

(defun xrplacax (n a)                               ;no error checking
  (arset memspace a n tdatum))

(defun xrplacdx (n b)
  (arset memspace b (1+ n) tdatum))

(defun xatom (a)
  (progl (setq temp free)
         (incr free 2)                               ;note space used
         (decr freecnt 2)
         (settype memspace temp tatom)
         (clrtype memspace temp tlist)
         (clrtype memspace temp tthread)
         (clrtype memspace temp trelocate)
         (xrplacax temp a)
         (xrplacdx temp '**atom**)))

(defun xatomp (n) (or (xnull n) (typeset? memspace n tatom)))
(defun xlistp (n) (or (xnull n) (typeset? memspace n tlist)))
(defun xobj? (n) (and (numberp n) (or (xatomp n) (xlistp n))))

(defun xdatum (n) (arref memspace n tdatum))
(defun setdatum (n d) (arset memspace d n tdatum))

(defun xgetatom (n) (cond ((xatomp n) (arref memspace n tdatum))
                          (t (error "in xgetatom -- not xatom" n))))

(defun xlist (a b) (xconsx a (xconsx b xn11)))

(setq xn11 0)
(defun xnull (a) (eq a xn11))

(defun xsetq (var val)
  (setq root (xconsx (xconsx var val) root)))

```

```

;
;;macros
;
(defmacro decr (var &optional (val 1))
  (cond ((eq val 1) '(setq ,var (1- ,var)))
        (t '(setq ,var (- ,var ,val)))))

(defmacro incr (var &optional (val 1))
  (cond ((eq val 1) '(setq ,var (1+ ,var)))
        (t '(setq ,var (+ ,var ,val)))))

(defmacro untildo (test body &optional (ans nil))
  '(do () (.test ,ans) ,body))

(defmacro settype (space n type &optional (val t))
  '(arset ,space ,val ,n ,type))

(defmacro typeset? (space n type &optional (val t))
  '(and (numberp ,n)
        (eq ,val (arref ,space ,n ,type))))

(defmacro defspace (space &rest typelist)
  '(progn (putprop ',space ,(length typelist) 'numtype)
          (putprop ',space ',typelist 'typelist)
          (setq ,@(do ((i 1 (1+ i))
                      (tlist (cdr typelist) (cdr tlist))
                      (tlist (list (car typelist) 0)
                                   (cons (car tlist) (cons i tlist))))
              ((= i (length typelist)) tlist)
            ())))

(defmacro numtypes (space) '(get ',space 'numtype))
(defmacro numwords (space) '(get ',space 'numword))
(defmacro types (space) '(get ',space 'typelist))

(defmacro clrtype (space n type)
  '(settype ,space ,n ,type nil))

(defmacro clrarray (space)
  '(do ((k1 0 (1+ k1)))
      ((= k1 (numwords ,space)) k1)
    (arset ,space 0 k1 0)))

(defmacro clrword (space n &optional size)
  '(do ((k 0 (1+ k)))
      ((= k ,(cond ((null size) '(numtypes ,space))
                  (t size))) ,n)
    (clrtype ,space ,n k)))

(defmacro copyword (space1 from space2 to)
  '(do ((k 0 (1+ k)))
      ((= k (numtypes ,space1)) ,to)
    (arset ,space2 (arref ,space1 ,from k) ,to k)))

(defmacro global (var)
  '(setq globlist (cons ,var globlist)))

(setq globlist nil)

(defmacro arref (arr . inds)
  '(arraycall t ,arr ,@inds))

(defmacro arset (arr value . inds)
  '(store (arraycall t ,arr ,@inds) ,value))

(defmacro defarray (name . dim)
  '(setq ,name (*array nil t ,@dim)))

```

```

:
: display fcns
:
(defun printsp (space)
  '(progn
    (print ,space)
    (print (types ,space))
    (do ((i 0 (1+ i)))
      ((= i (numwords ,space)) ())
      (print 'word) (princ i) (princ ": ")
      (do ((j 0 (1+ j)))
        ((= j (numtypes ,space)) ())
        (princ (arref ,space i j))
        (princ " ")))
      (print 'free) (princ "= ") (princ free)
      (print 'reespace) (princ "= ") (princ freecnt)
      (print 'root) (princ "= ") (princ root))))))

(defun prspace (space) '(printsp ,space)) ;alias
(defun prsp (space) '(printsp ,space)) ;alias

```

```

:
: these output the list structure (w/ brackets: [])
:

```

```

(defun xdisp (node)
  (cond ((xnull node) (princ "[ ]") ())
        ((xatom node) (princ (xgetatom node)))
        ((xlistp node)
         (princ "["
          (xdisp (xcar node))
          (princ " ")
          (xdisp2 (xcdr node))))))

(defun xdisp2 (node)
  (cond ((xnull node) (princ "]") ())
        ((xatom node)
         (princ ". ")
         (princ (xgetatom node))
         (princ "] ") ;for dotted pairs
        ((xlistp node)
         (xdisp (xcar node))
         (princ " ")
         (xdisp2 (xcdr node))))))

```

## References

1. A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. H. Baker, Jr. Actor Systems for Real-Time Computation. Tech. Rep. MIT LCS Report TR-197, Massachusetts Institute of Technology, March, 1978.
3. D. Bobrow. A Note on Hash Linking. *CACM* 18, 7 (July 1975).
4. W. Clinger. Foundations of Actor Semantics. Tech. Rep. AI-TR-633, Massachusetts Institute of Technology, May, 1981.
5. J. Cohen. Garbage Collection of Linked Data Structures. *Computing Surveys* 13, 3 (September 1981).
6. J. Carter and M. Wegman. Universal Classes of Hash Functions. *JCSS* 18 (1979).
7. L.P. Deutsch and D. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *CACM* 19, 9 (September 1978).
8. E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *CACM* 21, 11 (November 1978).
9. R. Fabry. Capability-Based Addressing. *CACM* 17, 7 (July 1974).
10. E. Feigenbaum. The Simulation of Verbal Learning Behavior. In *Computers and Thought*, E. Feigenbaum and E. Feldman, Ed., McGraw-Hill, New York, 1963.
11. J. Feldman and P. Rovner. An ALGOL-Based Associative Language. *CACM* 12, 8 (August 1969).
12. R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible Hashing -- A Fast Access Method for Dynamic Files. *ACM Trans. On DB Sys.* 4, 3 (September 1979).
13. E. Fredkin. Trie Memory. *CACM* 3, 7 (July 1960).
14. R. Fenichel and J. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *CACM* 12, 11 (November 1969).
15. G. Gonnet. Expected Length of the Longest Probe Sequence In Hash Code Searching. *JACM* 28, 2 (April 1981).
16. E. Goto. Monocopy and Associative Algorithms in an Extended Lisp. Information Science Laboratory Technical Report 74-03, University of Tokyo, 1974.
17. L. Guibas and R. Sedgwick. A Dichromatic Framework for Balanced Trees. Proc. 19th FOCS, October, 1978.
18. J. Holloway, G. Steele, G. Sussman, A. Bell. The SCHEME-79 Chip. MIT AI Lab Memo 559, Massachusetts Institute of Technology, January, 1980.



19. D. Knuth. *The Art of Computer Programming. Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1975.
20. D. Knuth. *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1975.
21. T. Kohonen. *Content-Addressable Memories*. Springer-Verlag, New York, 1980.
22. H. Lieberman and C. Hewitt. A Real Time Garbage Collector That Can Recover Temporary Storage Quickly. MIT AI Lab Memo 569, Massachusetts Institute of Technology, April, 1980.
23. M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms*. Elsevier North-Holland, New York, 1978.
24. G. Markowsky, J. Carter, and M. Wegman. Analysis of a Universal Class of Hash Functions. Proc. 7th Symp. on Mathematical Foundations of Computer Science, 1978.
25. M. Minsky. A LISP Garbage Collector Algorithm Using Serial Secondary Storage. MIT AI Lab Memo 58, Massachusetts Institute of Technology, October, 1963.
26. M. Minsky. A Framework for Representing Knowledge. MIT AI Lab Memo 306, Massachusetts Institute of Technology, June, 1974.
27. F. Morris. A Time- and Space-Efficient Garbage Compaction Algorithm. *CACM* 21, 8 (August 1978).
28. R. Pinter. New Hashing Schemes and Classes of Functions They Use. Unpublished Note. 1981
29. R. Rivest. Partial-Match Retrieval Algorithms. *SIAM J. Computing* 5, 1 (March 1976).
30. R. Roberts and I. Goldstein. The FRL Primer. MIT AI Lab Memo 408, Massachusetts Institute of Technology, July, 1977.
31. E. Sandewall. Ideas about Management of Data Bases. MIT AI Lab Memo 332, Massachusetts Institute of Technology, May, 1975.
32. D. Sarwate. A Note on Universal Classes of Hash Functions. *Info. Proc. Letters* 10, 1 (February 1980).
33. A. Schonage. Storage Modification Machines. *SIAM J. Computing* 9, 3 (August 1980).
34. T. Standish. *Data Structure Techniques*. Addison-Wesley, Reading, Mass., 1980.
35. G. Steele. Multiprocessing, Compactifying Garbage Collection. *CACM* 18, 9 (September 1975).
36. G. Steele. Destructive Reordering of CDR-Coded Lists. MIT AI Lab Memo 587, Massachusetts Institute of Technology, August, 1980.

37. G. Steele and G. Sussman. The Revised Report on SCHEME a Dialect of LISP. MIT AI Lab Memo 452, Massachusetts Institute of Technology, January, 1978.
38. P. Szolovits and W. Martin. Brand X Manual. Massachusetts Institute of Technology, November, 1980.
39. L. Thorelli. A Fast Compactifying Garbage Collector. *BIT 16* (1976).
40. J. Ullman. A Note on the Efficiency of Hash Functions. *JACM 19*, 3 (July 1972).
41. J. Ullman. *Principles of Database Systems*. Computer Science Press, Potomac, Md., 1980.
42. M. Wegman and J. Carter. New Classes and Applications of Hash Functions. Proc. 20th FOCS, October, 1979.
43. A. Yao. On Random 2-3 Trees. *Acta Informatica 9* (1978).
44. C. Zaniolo. Incomplete Database Information and Null Values: An Overview. Unpublished Memo, Bell Labs, Holmdel, N.J. 1981