1975 ACM Student Award
Paper: First Place

Multiprocessing Compactifying Garbage Collection

Guy L. Steele Jr.
Harvard University

Algorithms for a multiprocessing compactifying garbage collector are presented and discussed. The simple case of two processors, one performing LISP-like list operations and the other performing garbage collection continuously, is thoroughly examined. The necessary capabilities of each processor are defined, as well as interprocessor communication and interlocks. Complete procedures for garbage collection and for standard list processing primitives are presented and thoroughly explained. Particular attention is given to the problems of marking and relocating list cells while another processor may be operating on them. The primary aim throughout is to allow the list processor to run unimpeded while the other processor reclaims list storage The more complex cases involving several list processors and one or more garbage collection processors are also briefly discussed.

Key Words and Phrases: garbage collection, storage reclamation, reclaimer, storage allocation, multiprocessing, synchronization, semaphores, parallel processing, compactification, relocation, LISP, list processing, free storage, pointers, data structures, gc processor
CR Categories: 4.19, 4.32, 4.40, 4.49, 4.9

This paper was awarded First Place in ACM's 1975 George E. Forsythe Student Paper Competition.

Author's address: 410 Washington Street, Brighton, MA 02135.

1. Introduction

One of the major problems of list processing programs is that of reclaiming discarded list cells. As both Schorr and Waite [18] and Knuth [10, pp. 412–413] point out, there are three primary techniques for dealing with this problem.

The first is to place the responsibility of reclamation on the programmer. This technique was used in earlier general list processing systems [16]. Today, however, it is used primarily in small, special purpose applications; it is too burdensome and too error-prone for general purpose systems.

The second technique is for the list manipulation primitives to maintain a reference count for each cell indicating the number of other cells which point to it. This technique is best known for its use by Weizenbaum in SLIP [23, 24]. Whenever the reference count for a cell reaches zero, that cell is reclaimed automatically. This technique suffers three disadvantages. It requires room in each cell for a counter, which for small cells may cost 25 percent extra memory space or more. Furthermore, it is not guaranteed to reclaim circular (self-referent) list structures [11, 18]. Finally, the basic list processing primitives which create objects and copy pointers must constantly spend time updating reference counts.

The third technique is that of garbage collection, originally proposed by McCarthy [12] and used in the LISP 1.5 system [13]. Under this scheme the entire problem of storage reclamation is ignored until the list of available cells (the *freelist*) is exhausted. The list processing program is then temporarily suspended while a "garbage collector" routine determines which cells are no longer accessible to the program; these cells are subsequently returned to the freelist. This technique is guaranteed to reclaim circular structures, and typically requires only an extra bit or two for each cell. It also frees not only the high-level programmer, but also most of the list processing primitives, from concern for reclamation; only the primitives which construct new objects from available cells need invoke the garbage collector. The garbage collector itself is usually a self-contained routine, relatively disjoint from the rest of the list processing system.

Because of the advantages of the garbage collection method, particularly its generality and modularity, it has received the most attention in the past decade, and it is used in most current large list processing systems, including MACLISP [15], ECL [3], and INTERLISP [19]. Various refinements have been made over the years, but the basic garbage collection algorithm remains the same as it was ten years ago [8, 17]:

(1) (*Mark*) Determine which cells are still accessible to the list-processing program.
(2) (*Relocate*) Compact available cells into a contiguous region.
(3) (*Update*) Update all pointer references to relocated cells.

(4) (*Reclaim*) Collect inaccessible cells into a list; this becomes the new freelist.

Steps (2), (3), and (4) may also be performed in the order (4), (2), (3) [8]; furthermore, steps (2) and (3) may simply be omitted if compactification is not desired.

There is one primary disadvantage to the garbage collection method, however; this is the fact that list processing must be suspended periodically. The time at which a garbage collection will occur is relatively unpredictable to the list processor, particularly if the garbage collector performs sophisticated dynamic storage allocation. This unpredictability is directly related to the fact that the garbage collection process is relatively disjoint from the list processing itself. As list processing databases grow larger and larger, garbage collection takes longer and longer to perform. This is of little consequence in a batch situation, but many large list processing systems, such as algebraic manipulation systems and intelligent robotics programs, are meant for interactive or real-time use. In such situations long pauses for garbage collection may be annoying if not intolerable. As an example, an average MACSYMA [2] job running interactively in a MacLISP on a Digital Equipment Corp. PDP-10 computer [4] contains approximately 50,000 to 70,000 36-bit words of list data, and a garbage collection typically takes 1500 to 3000 msec of run time (about two to five times that much real time under daytime time-sharing loads). Similarly, Conrad [3] notes that an ECL program with 80,000 36-bit words of data may take 3000 to 6000 msec of runtime for a garbage collection. It is difficult for an interactive or real-time list processing program to provide adequate service when it must frequently suspend operations for several seconds in order to garbage collect. This problem grows increasingly worse as large interactive list processing systems grow ever larger.

This problem can be alleviated by building faster processors, of course, but this will go only so far. Even if a garbage collector is microprogrammed into the list processing computer it will still have to suspend list processing operations for some interval while garbage collection takes place. If the garbage collection technique is to be retained, then the only way to avoid this suspension of operations is to introduce parallelism; that is, to garbage collect while list operations are going on. (Knuth credits this idea to M. Minsky [10, exercise 2.3.5–12].) Note that parallelism does not necessarily imply simultaneity; that is, garbage collection could occur, for example, during keyboard input, as long as the garbage collection could be suspended on short notice in order to perform list processing on the input and later resumed without losing all the previous expended effort. The simplest case would be to time-share one processor between list processing and garbage collection, doing the garbage collection during keyboard input or at intervals scheduled by a clock. This would alleviate or eliminate the "embarrassing pause" problem, but would afford no net speedup; indeed it would introduce the overhead of context switching. Another method would be to have two processors, one for each purpose; in this case list processing and garbage collection could be simultaneous, with each running continuously. More complex arrangements could be imagined, such as one in which processors are allocated dynamically to garbage collection or to list processing as real-time needs dictate.

Parallel garbage collection could also be practical even if the spreading-out effect is not required. Typical large LISP jobs may spend from 10 to 40 percent of their time in garbage collection; running garbage collection in parallel could cut the total real time for a given task by close to this amount, without requiring the user to plan explicitly for parallelism. (Since the costs of CPUs have been dropping steadily, it would be practical to devote one processor to garbage collection even if it would be idle part of the time, as the above figures imply. Again, dynamic processor allocation could be used instead.)

Unfortunately the concept of parallel garbage collection raises yet more problems. The garbage collection processor can not assume that the list data will stay put while it determines which cells are no longer accessible; it must reclaim discarded cells even while the list processor uses them up again. Furthermore, if it is desired that the garbage collector relocate and compactify the list data (e.g. to improve virtual memory swapping performance), then even more serious synchronization problems must be solved, for a garbage collection processor cannot simply copy a cell elsewhere and reclaim it while a list processor may be operating on it.

Here we will concern ourselves with only the simple case of one list processor and one garbage collection processor. The necessary algorithms for both compactifying and noncompactifying garbage collection will be developed, as well as necessary modifications to the standard list processing primitives. The synchronization issues will be developed in detail. (Note: the various algorithms will be expressed in a language which is essentially Algol-like, but which contains the modes and data structures of the ECL language [21, 22]. Furthermore, LISP terminology will be used for many of the standard list processing concepts and operations. This strange combination is intended to maximize the product of readability, consistency, and convenience for the reader, since no one standard language currently embodies all the necessary concepts in a convenient form. An explanation of features borrowed from ECL is in the Appendix.)

2. The Database

Before we can proceed to a discussion of the various algorithms we must define the form of the data on which the processors are to operate. Informally, this is to

consist of a collection of list cells, as in LISP, each of which may contain pointers to other cells, organized into sets called spaces; we will also need pushdown lists for temporary storage and recursion, and semaphores for interprocessor communication.

More formally, let us define a *space* to be an ordered sequence of objects. Each object within a given space is the same in form as any other object, i.e. spaces are homogeneous. (The case of inhomogeneous spaces, or *heaps*, will be dealt with briefly in Section 7.) A space also has associated with it two pointers, called *freelist* and *lastfree*, which respectively point to the first and last objects in a linked chain of available cells within that space. The *freesem* and *sweepindex* components are used for synchronization and will be discussed later.[1]

space :: **ptr**(**struct**(*freesem: semaphore, freelist: pointer, sweepindex: int; lastfree: pointer, cells:* **seq**(*object*)));

Let semaphores be as defined by Dijkstra [5], for communication between the list processor and the garbage collection processor, and let *semaphore* be a primitive data type.

Let an *object* be a structure containing three components: an ordered sequence of pointers, and two bits called the *mark* bit and the *flag* bit. (In practice an object might contain other data as well, such as integers or characters, but we shall ignore that consideration here.)

object :: **struct**(*ptrs:* **seq**(*pointer*), *mark: bool, flag: bool*);

Let a *pointer* be a structure containing two components: a space indicator and an integer which must be a valid index within that space's sequence of objects. Thus a pointer identifies a particular object within a space. Note that from the pointer one can also identify the space itself.

pointer :: **struct**(*spc:* **oneof**(*space, pdl*), *adr: int*);

Let *nil* be a distinguished pointer which points to no actual object, and which by convention is used to terminate linked lists.

It will be convenient to define two functions for manipulating pointers, one to create them and one to follow them. The function *address*, given a space and an index within that space, creates a pointer to the object within the space with the specified index. The function *contents*, given a pointer, "returns" the object specified by the pointer. This quantity can be subscripted to select components of the object; e.g. *contents*(*x*).*ptrs*[*1*] takes a pointer *x* and yields the pointer contained in the first component of the specified object.

contents ← **expr**(*x: pointer*; **oneof**(*object, pdlobject*)) (*x.spc.cells*[*x.adr*]);

address ← **expr**(*s:* **oneof**(*space, pdl*), *j: int*; *pointer*) (**const**(*pointer* **of** *s, j*));

Let a *pdl* (pushdown list, or stack) be a structure containing a sequence of pdlobjects, and an integer called the *pdl index* which must be a valid index within the sequence. A *pdlobject* in turn is a structure containing a pointer and a *pdlmark* bit. This is intended to work as the usual stack; subroutines for pushing things onto the *pdl* and popping them off will be defined later. For some purposes a pdl is a kind of space; for example, a pointer may point to a pdlobject rather than an ordinary object. The *pdlsem* component is used for synchronization and will be discussed later.

pdl :: **ptr**(**struct**(*pdlsem: semaphore, index: int, gcdone: bool, cells:* **seq**(*pdlobject*)));

pdlobject :: **struct**(*ptr: pointer, pdlmark: bool*);

An object is said to be *accessible* if one of the following three conditions holds for it:

(1) It is in a distinguished space called *rootspace*. This is intended to correspond to the notion of "special value cells" or the "oblist" in LISP; that is, *rootspace* is a set of cells containing pointers to all directly accessible objects, such as constant structures and values of variables.

(2) A pointer to it is in the *j*th cell of a distinguished pdl called the *listpdl*, and the pdl index of the listpdl is greater than or equal to *j*. These represent temporary results of computations and values of local variables.

(3) A pointer to it is in some other accessible object. Thus it is accessible because a chain of pointers extends to it from an object directly accessible by virtue of conditions (1) or (2).

3. The Processors

The database is operated upon by two processors, the *list processor* and the *garbage collection processor* (hereafter called the *gc processor*). Each processor has a pdl associated with it, called respectively the *listpdl* and the *gcpdl*. The listpdl is used by the list processor for manipulation of list structure, for temporary variables (such as local variables for LISP functions), and for passing of arguments to functions. The gcpdl is used by the gc processor for the recursive tracing and marking of accessible list structure.

Each processor may perform almost any computation unrelated to the list cell database (e.g. arithmetic computation); each is presumed able to handle looping and recursive control structures and recursive passing of nonpointer arguments. Each has internal registers which may be used as temporaries; neither processor can

[1] The use of the **ptr** construct in the definition of *space* indicates that a *space* is really an ECL pointer to a space structure (see the Appendix). Thus other structures containing a *space* will not contain a copy of the entire space structure, but only the "address" of the space structure, so to speak. This kind of **ptr** is not to be confused with the *pointer* data structure defined below.

examine the other's registers, however, and thus the list processor may not assume that an object is safe from reclamation just because one of its internal registers contains a pointer to that object.

All spaces, pdls, objects, pointers, and semaphores can be operated on by either processor. We shall also assume that in the description of the algorithms any variables which are declared to be local are accessible only to the processor executing the algorithm, but that global variables are accessible to either processor.

In order for the two processors to cooperate, some restrictions must be placed on their operation. Under normal conditions the list processor will access and modify only accessible cells, and will only modify the mark and flag bits under special conditions. It will be assumed for each processor that examination of or assignment to a simple datum is an operation relatively continuous (i.e. indivisible) with respect to the other processor, where a *simple datum* is one of type *int*, *bool*, or *symbol*.[2] Examination of or assignment to a more complex object is not relatively continuous with respect to the other processor and therefore must be done circumspectly. All other operations are assumed to be totally asynchronous unless specially synchronized.

For synchronization purposes we shall assume the **P** and **V** semaphore primitives defined by Dijkstra [5]. Furthermore it will be convenient to define a pair of more complex synchronization operators in terms of the **P** and **V** primitives. These will be used to give a processor exclusive access to a single object in some space without having to lock the entire space or having to associate a semaphore with each object.

```
munch ← expr(x: pointer)
  begin
    P(munchsemaphore);
    while x = munchreg[him] do nothing;
    munchreg[me] ← x;
    V(munchsemaphore);
  end;

unmunch ← expr()
  munchreg[me] ← nullmunchpointer;
```

The pseudovariable *me* refers to the processor executing the function, and *him* to the other processor. The idea here is that each processor has a global "munch register" which contains a pointer to the object to which the processor desires exclusive access. If the other processor tries to *munch* the object before the first processor *unmunches* it, then the other processor will loop (or hang) until it can *munch* it.[3] (In practice such a mechanism could be implemented with two hardware registers and some simple logic in a manner which would cost little time unless a conflict occurred, a fairly infrequent event, one would hope.) The usefulness of these operators will become more apparent in Sections 5 and 6.

[2] The algorithms described later will depend quite heavily on this implicit synchronization function on variable access. This is not quite "nice" in some formal sense, and is not usually done in the literature on synchronization. In practice, however, the memories of multiprocessing systems do have this characteristic, and the purpose of this paper is to indicate a practical method of implementation. It would not be difficult to "clean up" the code by introducing extra semaphores.

[3] The *munch* operator as written above particularly depends on the implicit access synchronization mentioned before.

4. Techniques and Difficulties

At this point we must consider more fully the difficulties which arise when list processing is concurrent with garbage collection. The most serious of these difficulties is that of relocating objects within a space when the list processor may be operating on those objects. In order to solve this we adopt the following convention:

> If an accessible object's flag bit is *true*, then that object has been relocated, and the first pointer component of the object contains the new address of the object.

That is, the flag bit serves as an indirection marker. This is the key principle which allows the garbage collector to relocate cells under the feet, so to speak, of the list processor. This condition will be true only during the relocate and update phases; but when the gc processor is in these phases, then the list processor must be aware of this fact. During the relocate and update phases, if the list processor is about to operate on an object which may have been relocated, it must first check the flag bit of that object and fetch its first pointer component if the flag bit is true. We shall call this operation *normalization* of a pointer, and introduce a standard function for the purpose:

```
normalize ← expr(x: pointer; pointer)
  begin
    comment Any function which calls normalize must first perform a P(gcstatesem) ;
    if (gcstate = "relocate" or gcstate = "update")
        and contents(x).flag
      then contents(x).ptrs[1]
    else x;
  end;
```

These conventions handle the static case where an object has already been relocated. However, it does not cover all possible occurrences of dynamic interference during the relocation of the cell itself. In order to interlock the processors correctly, it is necessary for the relocation phase to *munch* an object before relocating it, and for the list processor to *munch* it before accessing or modifying its components. Furthermore, it is necessary to keep the garbage collector from shifting from one phase to another while the list processor is trying to determine what phase it is in. In order to do this we introduce two global variables, *gcstatesem* and *gcstate*. The former is a semaphore controlling access to the latter, which in turn has as its value one of the symbols *"mark"*, *"relocate"*, *"update"*, *"reclaim"*, or *"?"*. The

gc processor generally alters the value of *gcstate* only through the function *setgcstate*.

```
setgcstate ← expr(state: symbol)
  begin
    P(gcstatesem);
    gcstate ← state;
    V(gcstatesem);
  end;
```

Thus the list processor need only execute **P**(*gcstatesem*) in order to keep the gc processor from changing phases.

Another difficult problem is that of creating new objects from a freelist of available cells. The problem is twofold: first, as new objects are created, the mark phase must know that they are now accessible; second, during the reclaim phase, discarded cells must be added to the freelist without interfering for any length of time with the ability of the list processor to use the freelist for object creation. The solution to the first part is that the list processor, when creating a new object, must be aware of whether the gc processor is in the mark phase, and if so, ensure that correct action is taken. The solution to the second part is to use semaphores to interlock access to freelists, and to arrange for a freelist to be locked as little as possible. This is easier if we keep a pointer to the last cell of the freelist and have the gc processor append new cells to the end of the freelist while the list processor extracts cells from the beginning. In this way a freelist need only be interlocked with a semaphore if reduced to one cell in length, which would happen very seldom in practice.

Another difficulty arises if the list processor is permitted to arbitrarily modify pointer components of accessible objects to point to other accessible objects. This problem has many aspects. For example, suppose that an object has already been marked, and then its second pointer component is altered to point to an unmarked object, and finally all pointers to the unmarked object except the one in the marked object are erased. Unless special action is taken, the unmarked object will remain unmarked, and will be reclaimed as a discarded cell even though it is accessible. Therefore we impose another key rule:

> If the list processor, during the garbage collector's mark phase, replaces a component of a marked object with a pointer to an unmarked object, then it must ensure that the garbage collector reexamines the marked object.

Similarly, suppose that the gc processor is about to update a pointer in object *a* which points to relocated object *b*, whose new location is *b'*. Then the following sequence of events could occur: (1) The gc processor fetches the pointer to *b* from object *a*; (2) The list processor modifies the component pointing at *b* to point to object *d*; (3) The gc processor, in trying to normalize the component, modifies it to point to *b'*. Thus the component modification by the list processor could be lost totally. In this situation the *munch* operator must be used by both processors to interlock modifications to object *a*.

Table I. Meanings of Mark and Flag Bits

Mark bit / Flag bit	*false* / *false*	*false* / *true*	*true* / *false*	*true* / *true*
Mark phase	Cell not yet seen by mark and trace routine.	(Does not occur during mark phase.)	Cell seen by mark and trace routine. Cell is therefore accessible.	Cell on freelist. Should not be seen by mark and trace routine.
Relocate phase	Discarded cell. May be used to relocate an accessible object into if necessary.	Relocated cell. First pointer component indicates new location.	Accessible cell. May be relocated into new place if necessary.	Cell on freelist. Ignored by relocate phase.
Update phase	Discarded cell. Ignored by update phase.	Relocated cell. Ignored by update phase.	Accessible cell. Pointer components may need to be normalized.	Cell on freelist. Ignored by update phase.
Reclaim phase	Discarded cell. May be returned to freelist.	Relocated cell, now discarded. May be returned to freelist.	Accessible cell. Ignored by reclaim phase.	Cell on freelist. Ignored by reclaim phase.

A final synchronization problem is the use of the two pdls by each processor. Neither processor may push or pop things on a pdl while the other is doing so, for the pdl index could be inconsistent during such an operation. Furthermore, the list processor depends on the fact that items on the *listpdl* are accessible, and so the gc processor must be able to determine dynamically what objects are accessible from the *listpdl* even while the list processor is pushing and popping items. These problems can be resolved by associating a semaphore with each pdl, and using the fact that one can *munch* pdlobjects as well as ordinary objects.

5. The Garbage Collection Algorithms

The mark and flag bits determine the current status of a given object with respect to a phase of the garbage collector. In general, if both bits are *false*, then the object has been discarded and is therefore available. If the mark bit is *true*, then the object is accessible; if the flag bit is *true*, then the object has been relocated. The case where both mark and flag bits are *true* will mean that the object is on the freelist for its space; we may do this since accessible cells will never have both bits set at once. The most complex part of the garbage collection process is updating these bits correctly as the status of an object changes. Table I summarizes the meanings of these bits.

The basic structure of the garbage collection process is straightforward; the four phases are repeatedly exe-

cuted in sequence. If for some reason a noncompactifying garbage collection is desired (which reason could be computed on the basis of such factors as whether some freelist is getting dangerously short when the mark phase has just been completed), then the relocate and update phases can simply be skipped.

```
gc ← expr()
  while true do
    begin
      gcmark();
      if compactify then
        begin
          gcrelocate();
          gcupdate();
        end;
      gcreclaim();
    end;
```

At this point we need some primitives for manipulating the gcpdl, namely functions to push items onto the top and to pop them back off. The function *gcpush* receives an argument and pushes it; the function *gcpop* pops an item and returns it as its value. Each uses the semaphore *gcpdl.pdlsem* so that each processor may use the gcpdl without interference from the other.

```
gcpush ← expr(x: pointer)
  begin
    P(gcpdl.pdlsem);
    gcpdl.index ← gcpdl.index + 1;
    gcpdl.cells[gcpdl.index] ← x;
    V(gcpdl.pdlsem);
  end;

gcpop ← expr(; pointer)
  begin
    P(gcpdl.pdlsem);
    decl result: pointer byval gcpdl.cells[gcpdl.index].ptr;
    gcpdl.index ← gcpdl.index − 1;
    V(gcpdl.pdlsem);
    result;
  end;
```

The mark phase of the garbage collector (*gcmark*) uses a simple recursive trace-and-mark method for finding all accessible cells. (The Deutsch-Schorr-Waite method of using reversed pointers in the objects themselves as a stack [18; 10, p. 417] is not appropriate for use here because it would render the objects unusable to the list processor during the trace.[4]) The *gcmark* routine simply pushes a pointer to each object in *rootspace* in turn and calls a recursive trace routine (*gcmark1*) to trace out each list. All objects pointed to by the listpdl are also traced and marked; the *pdlsem* is used to keep the listpdl fixed while the gc processor determines whether or not all objects have been marked. The flag *listpdl.gcdone* is used to signal the list processor that the gc processor has finished marking from the listpdl, so that if pointers to other objects are subsequently put on the listpdl, the list processor will ensure that the gc processor will mark and trace them also, before the gc processor leaves the mark phase. The last loop in the mark phase ensures that everything has been marked, using *gcstatesem* to lock out the list processor; if the list processor has gotten an item onto the gcpdl, it must be marked and the test retried. (In practice this would happen a few times at most, because the list processor puts an item on the gcpdl only if it is unmarked, or, if it is a freshly created object, only if some component is unmarked [see the function *create* below]; but this is unlikely at the end of the mark phase.)

```
gcmark ← expr()
  begin
    setgcstate("mark");
    comment Mark from all objects in rootspace. ;
    for j from 1 to length(rootspace) do
      begin
        gcpush(address(rootspace, j));
        gcmark1();
      end;
    comment Mark from all listpdl slots in use. A semaphore is
      used because the number of slots in use is a function of time
      as the list processor pushes and pops things. The pdlmark
      for each slot is explicitly set to true rather than letting
      gcmark1 do it because it must be set before releasing the
      semaphore, and the semaphore must be released so as to
      block the list processor only minimally (see push).;
    for j from 1 until
                begin
                  P(listpdl.pdlsem);
                  j > listpdl.index;
                end
      do begin
           gcpush(listpdl.cells[j].ptr);
           listpdl.cells[j].pdlmark ← true;
           V(listpdl.pdlsem);
           gcmark1();
      end;
    comment Set a switch to tell the push function we are done
      marking from the listpdl. ;
    listpdl.gcdone ←true;
    V(listpdl.pdlsem);
    P(gcstatesem);
    comment We would like to terminate the mark phase, but the
      push or clobber functions may have pushed something. ;
    until gcpdl.index = 0 do
      begin
        V(gcstatesem);
        gcmark1();
        P(gcstatesem);
      end;
    comment It is safe to leave the mark phase now. However, we
      are not yet sure which phase we are going into (relocate or
      reclaim), so we will just temporarily let the state be "in-
      determinate." ;
    gcstate ← "?";
    listpdl.gcdone ← false;
    V(gcstatesem);
  end;


gcmark1 ← expr()
  while gcpdl.index > 0 do
```

[4] Richard Greenblatt has described to the author a hardware garbage collector for the PDP-6 once planned years ago, but unfortunately never built, at the MIT AI Lab, which would time-share with an applications program, and use the Deutsch-Schorr-Waite algorithm; the hardware would quickly unreverse all reversed pointers when switching back to list processing, then re-reverse them on resumption of garbage collection.

```
begin
  decl x: pointer byval gcpop();
  comment A pdlobject has to be handled a little bit specially. ;
  if x.spc = listpdl then
    begin
      contents(x).pdlmark ← true;
      x ← contents(x).ptr;
    end;
  comment Using the gcpdl as a stack, "recursively" mark objects accessible from the current one, unless the current one has been marked already. ;
  if not contents(x).mark then
    begin
      munch(x);
      for j from 1 to length(contents(x).ptrs) do
        gcpush(contents(x).ptrs[j]);
      contents(x).mark ← true;
      unmunch();
    end;
end;
```

The next phase of the garbage collector is responsible for relocating objects. As mentioned in Section 1, the motivation for relocation is primarily the reduction of the size of the working set in a virtual memory system. If one envisions LISP systems with addresses of 24 bits or more [7], relocation is absolutely necessary to prevent eventual thrashing. There are three main techniques which might be used. The first is the "copying" method, in which the garbage collector creates a second area for each space to be compactified, and copies all structure from the old area to the new one [14, 6]; this is used, for example, in the Multics implementation of MACLISP [15]. This technique could be used here, but is intended for large virtual memory systems, and tends to require much extra memory space for the copying. Furthermore, it is difficult, though not impossible, to adapt this technique to the present situation, in which new objects may need to be created while old ones are being relocated. The second method is the "sliding" method, in which the garbage collector slides all objects to one end of the space without altering their relative order [3, 20]; this technique is suited to applications involving heaps of odd sized objects, but will not do for our present purposes, because it requires the destruction of occupied cells before all pointers to them have been updated. The third technique is the "two-pointer" scheme, in which for each space the garbage collector uses two pointers, one sweeping up from the bottom and one down from the top. When the former reaches a discarded cell and the latter an accessible object, the object is relocated into the empty cell; the process terminates when the pointers meet [8; 17; 10, exercise 2.3.5-9]. Furthermore, this technique aids pointer updating because the evacuated cell can hold a pointer to the object's new location, which is precisely what we need; moreover the two-pointer scheme is simple to describe and implement. Therefore we will use the two-pointer technique here, recognizing that other schemes could possibly be adopted instead.

Originally the two-pointer scheme made freelists unnecessary, but we will find it necessary to retain the freelist so that the list processor can create new objects during the relocation phase. Thus our relocation technique here will never produce a completely contiguous storage region at any one instant of time, but will, we hope, keep the working set down to a reasonable size.

Just before an object is relocated to a new cell, its current location has its mark bit *true* and its flag bit *false*; these bits are each inverted to indicate the relocation. The new location has its mark bit set to *true* and its flag bit to *false* so that the reclaim phase will realize that the new location contains an accessible object. The first pointer component of the old location is altered to point to the new location. (Implementation note: in practice an object might contain no pointers, but might contain other data, such as integers or characters; as long as the object contained enough room for a pointer, the data in the old location could be overlaid with a pointer to the new location. Unfortunately this idea is difficult, if not impossible, to express in ECL and most other machine-independent high-level languages.) Note the use of the *munch* operator to interlock the alterations with respect to the list processor.

```
gcrelocate ← expr()
  begin
    setgcstate("relocate");
    relocate(space1);
    relocate(space2);
    ...
    relocate(spacen);
  end;
```

It is not necessary to relocate *rootspace*, since all objects in it are always marked anyway.

```
relocate ← expr(s: space)
  begin
    decl j: int byval 0;
    decl k: int byval length(s.cells) + 1;
    comment Sweep j from the low end and k from the high end until they meet in the middle somewhere. ;
    while (j ← j + 1) < (k ← k − 1) do
      begin
        comment Advance j upward to the lowest unmarked cell (or until k is reached). ;
        while j < k and s.cells[j].mark do
          j ← j + 1;
        comment Advance k downward to the highest marked cell (or until j is reached). ;
        until k ≤ j or (s.cells[k].mark and not s.cells[j].flag) do
          k ← k − 1;
        if j < k then
          begin
            comment Relocate an object. ;
            s.cells[j] ← s.cells[k];
            s.cells[j].mark ← true;
            munch(address(s, k));
            s.cells[k].mark ← false;
            s.cells[k].flag ← true;
            s.cells[k].ptrs[1] ← address(s, j);
            unmunch();
          end;
      end;
  end;
```

The update phase of the garbage collector is quite straightforward: it sweeps over each space, including *rootspace*, and normalizes all pointers in accessible objects. It must also normalize all pointers on the listpdl.

```
gcupdate ← expr()
  begin
    setgcstate("update");
    update(rootspace);
    update(space1);
    update(space2);
    ...
    update(spacen);
    pdlupdate();
  end;

update ← expr(s: space)
  begin
    for j from 1 to length(s.cells) do
      if s.cells[j].mark and not s.cells[j].flag then
        for k from 1 to length(s.cells[j].ptrs) do
          begin
            munch(address(s, j));
            if contents(s.cells[j].ptrs[k]).flag then
              s.cells[j].ptrs[k] ←
                contents(s.cells[j].ptrs[k]).ptrs[1];
            unmunch();
          end;
  end;

pdlupdate ← expr()
  begin
    for j from 1 until j > listpdl.index do
      begin
        munch(address(listpdl, j));
        if contents(listpdl.cells[j]. ptr).flag then
          listpdl.cells[j] ←
            contents(listpdl.cells[j]).ptrs[1];
        unmunch();
      end;
  end;
```

The reclaim phase of the garbage collector sweeps over all spaces except *rootspace*, searching for discarded objects (mark bit *false*); each has its mark and flag bits both set to *true* and then is appended to the end of the freelist for its space. Cells having the mark bit *true* and flag bit *false* (accessible cells) have their mark bits reset to *false* in preparation for the next garbage collection cycle. Cells with mark and flag bits both *true* are already on the freelist and so are ignored. All mark bits in *rootspace* and all *pdlmark* bits in the *listpdl* are also reset for the next gc cycle. The *sweepindex* component of the space is updated so that the *create* function can set the mark bit of a newly created object correctly. These components are initialized just before entering the reclaim phase proper (as reflected by *gcstate*).

```
gcreclaim ← expr()
  begin
    space1.sweepindex ← 0;
    space2.sweepindex ← 0;
    ...
    spacen.sweepindex ← 0;
    setgcstate("reclaim");
    reclaim(space1);
    reclaim(space2);
    ...
    reclaim(spacen);
    for j from 1 to length(rootspace.cells) do
      rootspace.cells[j].mark ← false;
    for j from 1 to length(listpdl.cells) do
      listpdl.cells[j].pdlmark ← false;
  end;

reclaim ← expr(s: space)
  for j from 1 to length(s.cells) do
    begin
      s.sweepindex ← j;
      munch(address(s, j));
      if not s.cells[j].mark then
      begin
        comment If the mark bit is false then the cell either is
          inaccessible (flag bit false) or has been relocated (flag
          bit true), and so we may reclaim it.  ;
        s.cells[j].mark ← true;
        s.cells[j].flag ← true;
        s.cells[j].ptrs[1] ← nil;
        P(s.freesem);
        if s.lastfree = nil then
          s.freelist ← address(s, j)
        else contents(s.lastfree).ptrs[1] ← address(s, j);
        s.lastfree ← address(s, j);
        V(s.freesem);
      end
      else if not s.cells[j].flag then
        s.cells[j].mark ← false;
      unmunch();
    end;
```

6. The List Processing Primitives

The list processor, in order to accomplish its work, requires a minimal set of primitives for list manipulation; the other operations may be defined in terms of this minimal set. We shall need primitives for creation of new objects from the list of available cells (the *cons* function of LISP), for selection of components of objects (*car* and *cdr*), for alteration of components (*rplaca* and *rplacd*), for determination of which space an object belongs to (*atom*), and for comparison of pointers for identity (*eq*). We shall also need operations on the listpdl, not only the standard pushing and popping functions but also some for random access; these will be helpful for defining composite functions in terms of the primitives.

The argument passing convention for list processing primitives should be familiar to those who have worked with LISP compilers or similar programs: pointer arguments are passed on the listpdl, with the last argument on top; a pointer result is returned on top of the listpdl. We will not concern ourselves with how nonpointer arguments or results are handled, but will leave that to the underlying host language (the examples below will make this clear).

The *push* and *pop* functions operate in the usual

manner on the listpdl, with some extra interlocks. The two main features of *push* are that it normalizes its argument (which is not passed on the pdl [!] but rather via the "host language mechanism," and so may be called only from some other primitive appropriately interlocked with the gc processor), using *munch* to prevent the gc processor from trying to update its contents; and that it explicitly pushes the address of the pdl cell onto the gcpdl if the gc processor is marking and has passed that point on the pdl. (This should not happen often, since the gc processor marks from the listpdl last.) Note that it pushes the address of the pdl cell, and not its pointer argument, since the argument is likely to be popped again soon and possibly discarded, whereas the pdl cell is likely to have something worth looking at when the gc processor gets around to examining it. There is a special check in *gcmark1* for this case.

```
push ← expr(x: pointer)
  begin
    P(listpdl.pdlsem);
    listpdl.index ← listpdl.index + 1;
    munch(address(listpdl, listpdl.index));
    listpdl.cells[listpdl.index].ptr ← normalize(x);
    unmunch();
    if gcstate = "mark" and
      listpdl.gcdone and
      listpdl.cells[listpdl.index].pdlmark and
      not contents(x).mark then
      begin
        listpdl.cells[listpdl.index].pdlmark ← false;
        gcpush(address(listpdl, listpdl.index));
      end;
    V(listpdl.pdlsem);
  end;
```

The *pop* function pops an item from the listpdl and returns it (via the host language return mechanism!).

```
pop ← expr(; pointer)
  begin
    P(listpdl.pdlsem);
    decl result: pointer byval
      listpdl.cells[listpdl.index].ptr;
    listpdl.index ← listpdl.index − 1;
    V(listpdl.pdlsem);
    result;
  end;
```

The *push* and *pop* functions may only be used by primitives which have already locked *gcstatesem*. For the benefit of programs which want to push and pop items, we define *pushitem* and *popitem*, which merely make calls to *push* and *pop* with locking of *gcstatesem* around them. Note that *popitem* merely throws the result of *pop* away, since there must be a *gcstatesem* interlock around the passing of "loose pointers" through the host language mechanism. By the same token, it is only safe to use *pushitem* on pointers to objects in *rootspace*, since only such objects are never relocated or reclaimed.

```
pushitem ← expr(x: pointer)
  begin
    P(gcstatesem);
    push(x);
    v(gcstatesem);
  end;

popitem ← expr()
  begin
    P(gcstatesem);
    pop();
    V(gcstatesem);
  end;
```

The generalized object creation function (*create*) receives a space *s* and a number *n*; there should be *n* pointer arguments on top of the listpdl, and objects in space *s* should each contain *n* pointer components. The *create* function extracts an available cell from the freelist of space *s*, resets the mark and flag bits appropriately, installs its pointer arguments as components of the new object, and returns a pointer to the new object on top of the listpdl.

```
create ← expr(s: space, n: int)
  begin
    while s.freelist = nil do nothing;
    P(gcstatesem);
    decl newcell: pointer byval s.freelist;
    decl sw: bool byval gcstate = "reclaim"
          and s.freelist = s.lastfree;
    if sw then P(s.freesem);
    s.freelist ← contents(newcell).ptrs[1];
    if s.freelist = nil then s. lastfree ← nil;
    if sw then V(s.freesem);
    munch(newcell);
    comment If the reclaim phase has swept past newcell, then
      we must initialize the mark bit to false instead of true. ;
    decl newmark: bool byval
          if gcstate = "reclaim"
            then s.sweepindex ≤ newcell.adr
            else true;
    comment The determination for the value for newmark here
      is primarily a heuristic to prevent a kind of thrashing near
      the end of the mark phase caused by pushing many newly
      created objects onto the listpdl. ;
    for j from n by −1 to 1 do
      begin
        decl x: pointer byval pop();
        if gcstate = "update" then x ← normalize(x);
        contents(newcell).ptrs[j] ← x;
        if gcstate = "mark" then
          newmark ← newmark and contents(x).mark;
      end;
    comment Munch the new cell while we change the mark and
      flag bits. ;
    contents(newcell).mark ← newmark;
    contents(newcell).flag ← false;
    unmunch();
    push(newcell);
    V(gcstatesem);
  end;
```

First, *create* checks the freelist for an available cell; if none is available, it loops until one is. (Instead of looping, it might try to allocate some storage dynamically.) Then, like most of the primitives we will define, *create* locks *gcstatesem* so the *gc* processor will not switch from one phase to another. (It must not lock *gcstatesem* until after checking the freelist, or it might lock the

garbage collector out of the reclaim phase! Furthermore, once the freelist is nonempty, it will remain nonempty until the list processor extracts a cell from it.) If there is only one object on the freelist it is necessary to lock the *freesem* for the space *s* if the gc processor is in its reclaim phase; if the freelist is longer than that, then the gc processor will only operate on the tail of the list, and therefore no interlocking is necessary. After removing the cell from the freelist, *create* installs the supplied pointers as the components of the new object, while calculating a boolean value *newmark*. This is a heuristic for the case when the gc processor is in the mark phase. If all the supplied pointers are already marked, then the gc processor is probably well into the mark phase, and the new cell probably will not be discarded before the mark phase finishes, so it might as well have its mark bit set to *true*, since doing so won't prevent *gcmark1* from seeing an unmarked but accessible object; but if some supplied pointer is not marked, then we may not immediately set the mark bit for the new object to *true*. Note that in the reclaim phase, the mark bit always being *true* prevents the reclaim phase from sweeping up the cell. Furthermore, since the bits are both *true* until all components have been installed, the update phase will not molest the new cell; the pointers are also normalized during the update phase, so that is no worry. During the reclaim phase, it is necessary to decide whether the sweep has passed the new cell or not, because if not, the mark bit must be set *true* so that the new object will not be reclaimed; and if so, it must be set *false* in preparation for the following mark phase. The *sweepindex* component of the space communicates the necessary information; the use of *munch* in both *create* and *reclaim* prevents the reclaim sweep from passing in the midst of object creation.

The function *select* takes a pointer on the listpdl and an integer *j*, and returns the *j*th component of the specified object on the listpdl. It is necessary to munch the argument during the relocate phase, so that the argument may be normalized and the component extracted without fear that the first component may be altered by relocation.

```
select ← expr(j: int)
  begin
    P(gcstatesem);
    decl x: pointer byval pop();
    if gcstate = "relocate" then munch(x);
    x ← normalize(contents(normalize(x)). ptrs[j]);
    if gcstate = "relocate" then unmunch();
    push(x);
    V(gcstatesem);
  end;
```

In order to modify components of objects, we have the primitive *clobber*. It takes two pointer arguments and an integer *j*; the *j*th pointer component of the object pointed to by the first argument is replaced by the second argument. The object being modified is *munched* during the modification itself. The only other problem occurs if the gc processor is marking, the first argument points to a marked object, and the second argument to an unmarked object; it is necessary to ensure that the gc processor eventually marks the second argument. The *clobber* function pulls a trick similar to that of *push*: it does not put its second argument onto the gcpdl, but rather its first argument, turning off the mark bit of the first argument! The rationale is similar: the object may be repeatedly modified, so that one may gain by doing it this way by avoiding many unnecessary pushes onto the gcpdl. In practice, one might distinguish two classes of objects, and *clobber* would push its first or its second argument onto the gcpdl as desired; value cells (cells holding values of LISP variables), for example, are likely to be clobbered repeatedly, but ordinary list cells not so often.

```
clobber ← expr(j: int)
  begin
    P(gcstatesem);
    decl y: pointer byval pop();
    decl x: pointer byval pop();
    if gcstate = "update" then y ← normalize(y);
    munch(x);
    contents(normalize(x)).ptrs[j] ← y;
    unmunch();
    if gcstate = "mark" and
      contents(x).mark and
      not contents(y).mark
      then begin
        contents(x).mark ← false;
        gcpush(x);
      end;
    V(gcstatesem);
  end;
```

Determination of which space an object belongs to is quite straightforward, since objects are never relocated from one space to another; the space information is in the pointer itself.

Comparison of pointers is fairly easy, but requires care in the case where the pointers are to the same object but one pointer is to the old location and the other to the relocated location. A *munch* operation guards against this possibility.

```
identical ← expr(; bool)
  begin
    P(gcstatesem);
    decl x: pointer byval pop();
    decl y: pointer byval pop();
    if x.spc = y.spc and x.adr ≠ y.adr then
      begin
        comment If indeed x and y are identical, then munching x also munches y during normalization. If not, we will return false no matter what the result of normalizing, and so it matters not that y is not munched. ;
        munch(x);
        x ← normalize(x);
        y ← normalize(y);
        unmunch();
      end;
    decl result: bool byval x.spc = y.spc and x.adr = y.adr;
    V(gcstatesem);
    result;
  end;
```

In order to define composite functions using the primitives defined thus far we will need two more primitives which will give us random access to pointers on the listpdl. Each takes an argument specifying which pointer to access. The argument, which should be nonpositive, is added to the pdl index to find the desired pointer; this allows specification of the pointer in terms of a displacement from the top of the pdl, generally the most convenient method. The function *pdlget* fetches such a pointer and pushes it onto the top of the listpdl; *pdlput* pops a pointer and puts it into the specified slot in the listpdl. There are careful checks in *pdlput* similar to those in *push* and *clobber* because of the pointer modification which occurs; *pdlget* simply uses *push*. Note that both functions calculate which pdl slot to examine or modify before performing the push or pop.

```
pdlget ← expr(j: int)
  begin
    P(gcstatesem);
    push(normalize(listpdl.cells[listpdl.index + j]ptr));
    V(gcstatesem);
  end;

pdlput ← expr(j: int)
  begin
    decl k: int byval listpdl.index + j;
    P(gcstatesem);
    decl x: pointer byval pop();
    munch(address(listpdl, k));
    if gcstate = "update" then x ← normalize(x);
    listpdl.cells[k]. ptr ← x;
    unmunch();
    if gcstate = "mark" and
      listpdl.cells[k].mark and
      not contents(x).mark
      then begin
             listpdl.cells[k].mark ← false;
             gcpush(address(listpdl, k));
           end;
    V(gcstatesem);
  end;
```

Now that we have defined a full set of list processing primitives, we may define, as an example, the standard LISP primitives. Let *listspace* be a space of objects which each contain two pointers, and define atomic objects to be all objects, and only those objects, not in *listspace*.

```
car ← expr()(select(1));
cdr ← expr()(select(2));
cons ← expr()(create (listspace, 2));
rplaca ← expr()(clobber(1));
rplacd ← expr()(clobber(2));
atomic ← expr(; bool)(pop().spc ≠ listspace);
atom ← expr()(push(if atomic() then t else nil));
eq ← expr()(push(if identical() then t else nil));
```

As a further example, here is a version of the LISP function *equal*, which recursively compares two list structures for equivalence. Two structures are considered equivalent if they are the same structure, or if they are both lists and their cars and cdrs are respectively equivalent. First the function *equal* is expressed in LISP, and then in terms of our primitives. The style of coding of the latter looks much like the output of a LISP compiler: at each step the arguments to a function are put on top of the listpdl, then the function is called. Local variables are kept on the listpdl as well; the purpose of *pdlget* and *pdlput* is to examine and set such local variables. (Also, like the output of a good LISP compiler, the latter version uses *identical* and *atomic* instead of *eq* and *atom*!)

```
(define equal
        (lambda (x y)
                (cond ((eq x y) t)
                      ((or (atom x) (atom y)) nil)
                      ((not (null (equal (car x) (car y))))
                       (equal (cdr x) (cdr y)))
                      (t nil))))

equal ← expr()
  comment We assume t and nil as directly accessible constants.
    Strictly speaking, they should be referenced as the contents of
    some cell in rootspace.  ;
  if begin pdlget(-1); pdlget(-1); identical() end
      then begin popitem(); popitem(); pushitem(t) end
  else if begin pdlget(-1); atomic() end
          or begin pdlget(0); atomic() end
      then begin popitem(); popitem(); pushitem(nil) end
  else if begin
            pdlget(-1);
            car();
            pdlget(-1);
            cdr();
            equal();
            pushitem(nil);
            not identical();
          end
    then begin
            pdlget(-1);
            cdr();
            pdlput(-2);
            cdr();
            equal();
          end
  else begin popitem(); popitem(); pushitem(nil) end;
```

7. Modifications and Future Work

The next step, of course, is to consider the interactions which would be necessary if more than two processors were involved. As long as there is only one gc processor involved, the synchronization interlocks would not need to be much more complex than they are in the algorithms presented here. The definitions of *munch* and *unmunch* would need to be extended, and the *create* function would need to lock out other list processors while manipulating the freelist. One pitfall is that if the list processors are allowed to push and pop each other's listpdls, then during a mark phase a pointer to an unmarked object might be repeatedly transferred from one pdl to another, pushing entries on the gcpdl, but

always being popped from a listpdl before the gc processor can examine the pdl slot, and so the object might never get marked. This kind of game probably would not last very long, but should be anticipated; perhaps in such a case a pointer to the unmarked object itself should be put on the gcpdl.

Graver difficulties arise if several gc processors are involved. Two possible schemes come to mind. The first is to have the gc processors always all be in the same phase, and to divide the work of each phase among them. During the mark phase each processor would grab something from the gcpdl, and use its own private pdl to trace and mark from that object, then examine the gcpdl again. If the gcpdl became empty, then the next processor to examine it would fetch the next item from *rootspace* or from a listpdl. During the other three phases, each processor would tackle one space apiece at a time until all spaces were processed. The other possible scheme, somewhat more complex, would be to assign certain spaces semipermanently to other processors, and let each processor collect garbage asynchronously. Coordinating the marking of pointers from one space to another could be quite challenging. Information about into which other spaces an object of a given space can point could be used to advantage here. In particular the "pure free storage" scheme could yield great savings in time; this involves splitting each space into two spaces, one of which, the "pure" one, is never garbage collected. Objects in pure spaces are constrained to point only at other pure objects; thus the gc processor need never mark or trace through pure objects. Since a large list processing system typically contains a large amount of static data, this can save half or more of the gc processor's effort. (This scheme has the further advantage that in a time-sharing situation pure list structure can be shared, via page mapping, between unrelated processes. It has been used with great success in the PDP-10 implementation of MacLISP [15], and a somewhat less efficient, though more general, variant is used in InterLISP [19].) Updating pointers with several gc processors would be somewhat less difficult; a gc processor could relocate a space, then issue a signal to all gc processors to update pointers in their spaces. When every other processor had done so and replied to that effect, then the first gc processor could carry on with its reclaim phase.

The ultimate extension of this scheme would be dynamic allocation of processors to garbage collection or to list processing. Clever heuristics would be needed to decide whether switching a processor to garbage collection at time t is desirable to forestall having to wait on an empty freelist at time $t + n$.

In the case of a single processor time-shared between list processing and garbage collection in order to alleviate the embarrassing pause problem, a specially microcoded processor could eliminates many of the interlocks described in the algorithms above merely by switching to garbage collection only on completion of a list processing primitive. Thus instead of assuming, as above, only that memory references were indivisible, one could have *create*, *clobber*, *gcpush*, etc. as indivisible primitives. The technique for the management of the mark and flag bits would still be relevant, of course. This would probably not be feasible on a general purpose processor, but would be ideal on the microcoded Greenblatt LISP machine [7, 9], for which it is indeed true that operation like *car*, *cons*, and *atom* are machine-language primitives.

In another direction, a desirable extension would be the handling of *heaps* containing objects of odd sizes. The two-pointer relocation scheme would not be applicable to this case, and the "sliding" method could not be used either, as mentioned in Section 5. The copying method would have to be used; this would require some "breathing room" to perform, but could be done.[5]

Until the algorithms presented here are actually implemented, it is difficult to predict what the performance gains will be. The gain would probably be relatively small if it were installed in standard existing LISPs on standard, general purpose computers like the PDP-10, because of the necessity of performing semaphore-type operations so frequently. The ideal situation would use specially designed processors, possibly microcoded, which would have the primitives defined in this paper as actual hardware primitives at some level; in particular, it would have some standard semaphores and *munch* registers built in, so that processor interlocks would be almost cost-free.

8. Conclusions

The algorithms presented here could be used to achieve the parallel processing of garbage collection, possibly yielding a net speedup, but more importantly eliminating the periodic suspension of list processing operations necessitated by standard garbage collection methods. It will probably be necessary to design special processors, possibly microcoded, to implement these techniques efficiently; as the prices of processors drop and microcoded processors become more common, this alternative becomes more feasible. If this is done, the list processor can run almost unimpeded, since necessary processor interlocking can be achieved through special hardware in such a way that the list processor almost never needs to wait on the gc processor, and then only for short periods of time.

[5] The copying method is also tricky because we require that the list processor be able to create new objects while the garbage collector is busy copying; should new objects be created in the copied-from space, the copied-to space, or a third space?

Appendix. ECL Language Features

The ECL language allows the use of user-defined data types (modes) and structures (with, in the author's opinion, much greater flexibility and readability than Algol 68). Certain features of the ECL language are used in this paper, sometimes with blatant disregard for certain technical issues; for a full description of these facilities the reader should consult Wegbreit et al. [21] or [22].

The mode definition operation is "::". The statement

foo :: ⟨mode⟩;

defines *foo* as the name of the specified mode.

Modes may be built up by using certain mode construction functions. The expression **seq**(⟨mode⟩) yields a new mode which is a sequence, or array, of the specified mode. This new mode has no implied upper bound on the subscript range; the upper bound is declared when a specific object is created. (In this paper the declaration of all such structures has been omitted as unnecessary to the understanding of the algorithms.) The function *length* of such an object will yield the maximum subscript for that object. The minimum subscript is 1. If *bar* is an object of mode **seq**(*int*), then the expression *bar*[*j*] yields the integer which is the *j*th component of the sequence.

The expression **oneof**(⟨mode1⟩, ⟨mode2⟩, . . . , ⟨mode*n*⟩) yields a mode such that an object of any of the specified modes may be considered to be of the new mode also. Any specific object satisfying this new mode will have one of the specific modes as its mode; i.e. when creating an object of mode **oneof**(*int*, *bool*) it must be an *int* or a *bool*. (For a precise explanation, see Wegbreit et al. [21] or [22].)

The **struct** operator is used to create structure modes:

foo :: **struct**(⟨name1⟩: ⟨mode1⟩, ⟨name2⟩: ⟨mode2⟩, . . . , ⟨name*n*⟩: ⟨mode*n*⟩);

An object of mode *foo* would be a structure with components named ⟨name1⟩ . . . ⟨name*n*⟩. Each component must be of its respectively specified mode. In order to select a component of a structure the operator "." is used. Thus if the object *quux* is of mode **struct**(*zip*: *int*, *zap*: **seq**(*bool*)) then *quux.zip* is an integer, *quux.zap* is a sequence of *bool*s, and *quux.zap*[3] is a *bool*. Note that "." is an operator and not just a syntactic trick; it may, for example, take a function call on its left, providing the function returns a structure with a component of the correct name: *randomfn*(*x*, *y*).*componentname* is quite legitimate. Provided the modes and objects involved are all correct, there is no reason not to intermix the "." and "[ ]" types of selection in one expression; one may think of them as being performed from left to right. Thus the expression:

usa.government.congress.senate[*43*].*children*[*2*].*name.first*

presumably would yield the first name of the forty-third senator's second child.

The expression **ptr**(⟨mode⟩) yields a mode which is a "pointer" to ⟨mode⟩. This allows structures to be shared between other structures, and allows passing of **ptrs** in place of entire large structures. For example, consider the following:

```
ztesch :: struct(this: frob, that: frob);
frob :: ptr(seq(int));
consfrob ← exp(fr: frob; int)
  begin
    decl zt: ztesch;
    zt.this ← fr;
    zt.that ← fr;
    zt.this[43] ← 27;
    zt.that[43];
  end;
```

Invoking *consfrob* with a *frob* should return the value 27, because *zt.this* and *zt.that* are **ptrs** to the same frob structure. The argument to *consfrob* has also been changed. If the mode *frob* had been defined without the **ptr** construct, then *zt* would contain two entire copies of *fr*, and the original argument would not have been altered; as it is, *zt* need only contain two copies of **ptrs** to *fr*. This can be important if a *frob* can contain a sequence of ten thousand integers!

Objects of a given mode can be constructed with the **const** construct. The general form is:

const(⟨mode⟩ **of** ⟨item1⟩, ⟨item2⟩, . . . , ⟨item*n*⟩)

This constructs a new object of mode ⟨mode⟩ (which should be a **struct**-type mode), with its components initialized to the specified items, and returns it.

Functions are constructed via the **expr** construct. This bears great similarity to the *lambda* construct of LISP, with the addition of mode specifications. The general form is:

expr(⟨var1⟩: ⟨mode1⟩, . . . , ⟨var*n*⟩: ⟨mode*n*⟩; ⟨returnmode⟩) ⟨expression⟩

If ⟨returnmode⟩ is omitted, then the function always returns *nothing* (a special object of mode *none*).

Local variables may be defined within **begin-end** blocks by using the **decl** construct. It may appear anywhere in a block; if it appears after some executable statements, then the variable is not declared, in effect, until the **decl** is reached in execution. The variable may be given an initial value by using the optional **byval** clause:

decl *temp*: *bool*;
decl *argrag*: *int* **byval** *f*(*x*) + *3*;

Acknowledgments. The author wishes to acknowledge the kind help of Patricia Griffiths, Richard M. Stallman, Richard Greenblatt, Richard L. Bryan, Gerald J. Sussman, Michael R. Genesereth, Stavros M. Macrakis, and the referees, who each provided thought-

ful criticism and invaluable advice. This paper stemmed originally from a discussion with Charles J. Prenner after a lecture in his course at Harvard University on control structures.

Received September 1974; revised January 1975

References

[Reference 1 is not cited in the text.]

1. Berkeley, E.C., and Bobrow, Daniel G. (Eds.) *The Programming Language LISP: Its Operation and Applications.* Information International, Inc., Cambridge, Mass., 1964.
2. Bogen, R.A. *MACSYMA Reference Manual.* Project MAC, Mathlab Group, MIT, Cambridge, Mass., 1974.
3. Conrad, W.R. A compactifying garbage collector for ECL's non-homogeneous heap. Tech. Rep. 2-74, Center for Research in Computing Technology, Harvard U., Cambridge, Mass., Feb. 1974.
4. Digital Equipment Corp. *DecSystem 10 Assembly Language Handbook (third ed.).* Maynard, Mass., 1973.
5. Dijkstra, E.W. The structure of "THE"-multiprogramming system. *Comm. ACM 11,* 5 (May 1968), 345.
6. Fenichel, R.R., and Yochelson, J.C. A LISP garbage collector for virtual-memory computer systems. *Comm. ACM 12,* 11 (Nov. 1969), 611–612.
7. Greenblatt, R. The LISP machine. MIT Artificial Intelligence Working Paper 79, MIT, Cambridge, Mass., Nov. 1974.
8. Hart, T.P., and Evans, T.G. Notes on implementing LISP for the M-460 computer. Ref. [1], 191–203.
9. Knight, T. The CONS microprocessor. MIT Artificial Intelligence Working Paper 80, MIT, Cambridge, Mass., Nov. 1974.
10. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
11. McBeth, J.H. On the reference counter method (letter). *Comm. ACM 6,* 9 (Sept. 1963), 575.
12. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, I. *Comm. ACM 3,* 4 (April 1960), 184–195.
13. McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.
14. Minsky, M.L. A LISP garbage collector using serial secondary storage. MIT Artificial Intelligence Memo No. 58 (revised), MIT Cambridge, Mass., Dec. 1963.
15. Moon, D.A. *MacLISP Reference Manual.* Project MAC, MIT, Cambridge, Mass., April 1974.
16. Newell, A. *Information Processing Language V Manual.* Prentice-Hall, Englewood Cliffs, N.J., 1961.
17. Saunders, R.A. The LISP system for the Q-32 computer. Ref. [1], 220–231.
18. Schorr, H., and Waite, W.M. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM 10,* 8 (Aug. 1967), 501–506.
19. Teitelman, W. *InterLISP Reference Manual.* Xerox Corp., Palo Alto, Calif., 1974, 3.11–3.15.
20. Wegbreit, B. A generalised compactifying garbage collector. *Computer J. 15,* 3 (Aug. 1972), 204–208.
21. Wegbreit, B., et al. ECL Programmer's Manual. Tech. Rep. 21–72, Center for Research in Computing Technology, Harvard U., Cambridge, Mass., Sept. 1972.
22. Wegbreit, B. The treatment of data types in EL1. *Comm. ACM 17,* 5 (May 1974), 251–264.
23. Weizenbaum, J. Knotted list structures. *Comm. ACM 5,* 3 (March 1962), 161–165.
24. Weizenbaum, J. Symmetric list processor. *Comm. ACM 6,* 10 (Sept. 1963), 524–544.