Programming Techniques

ROBERT MORRIS, Editor

Compact List Representation: Definition, Garbage Collection, and System Implementation

WILFRED J. HANSEN*
Stanford University,† Stanford, California

Compact lists are stored sequentially in memory, rather than chained with pointers. Since this is not always convenient, the Swym system permits a list to be chained, compact, or any combination of the two. A description is given of that list representation and the operators implemented (most are similar to those of LISP 1.5). The system garbage collector attempts to make all lists compact; it relocates and rearranges all of list storage using temporary storage. This unique list-compacting garbage collection algorithm is presented in detail. Several classes of the macros used to implement the system are described. Finally, consideration is given to those design factors essential to the success of a plex processing system implementation.

KEY WORDS AND PHRASES: data structure, data representation, list structure, list representation, list, compact list, garbage collection, relocation, storage reclamation, macro, primitive list operations, plex processing, plex, pointer, list processing system, LISP, free storage
CR CATEGORIES: 3.49, 4.20, 4.22, 4.49, 4.9

Introduction

A *plex*[1] is a collection of data items stored in one or more locations of memory. Manipulation of such collections as entities is called *plex processing* and is characterized by the following: (1) a data item may be a *pointer* (the address of a plex), and (2) plexes are created and discarded during execution. Plex processing can simplify programming and actually reduce memory requirements by permitting very flexible memory allocation. The design phase of any large programming project is incomplete without consideration of plexes as a form of data representation and plex processing as a programming technique.

Too frequently, computer science literature reports the capabilities of a system without reporting the implementation techniques, even though the latter are more likely to apply to the design of other systems. This situation has been improved for plex processing by D. Knuth; [3] provides excellent descriptions of numerous plex processing techniques. Good reviews of the literature and further references are in [10], [1], and [4]. Interesting projects are reported in [8] and [11], the latter being part of SDC's TM-3417 series describing the LISP 2 implementation for the IBM 360. The full Swym report [2] details the storage management alternatives for implementing the plex processing facilities of a higher-level language.

While investigating graphic applications of plex processing, the Stanford Graphics Project developed a list representation—called *compact lists*—that can reduce storage requirements by up to half. To experiment with this new representation, the Swym plex processing system was implemented. A report is given here on the techniques of that implementation and some experimental results. The focus is on the Swym garbage collector because it is a vital part of the system and employs a unique algorithm.

The difference between conventional list representations and compact lists parallels the difference between the IBM 650 and most other computers. 650 instructions had two address fields: one for the operand and one for the next instruction. Most other computers save memory by assuming that the instructions are sequential. When the instruction sequence is broken, a "branch" instruction continues execution elsewhere. Like the 650, many list representations use two pointers for each element of a list: one to the element and one to the rest of the list. On the other hand, list storage can be conserved by storing lists sequentially in memory; then only the pointers at the elements are required. But if that is the only way lists can be stored, certain list operations will be time-consuming. The Swym solution is to allow a "list branch" pointer. Lists are normally sequential, but when a list cannot be sequential, it is continued with a "list branch" pointer. Figure 1 illustrates several list structures in both the old and new representations. Note that a "list branch" pointer is called a **rst** pointer because it points to the rest of the list. In Section 1 a description is given of the details

The work reported here was supported in part by the National Science Foundation, Grant GP-7615.
* Present address: Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois.
† Department of Computer Science, Stanford Graphic Project.
[1] This term was coined by D. T. Ross [9], who used it to represent a collection of "*n*-component elements" connected by pointers. Common usage (and this paper) calls an *n*-component element a plex and a collection of these a plex structure. A list is one form of plex structure.

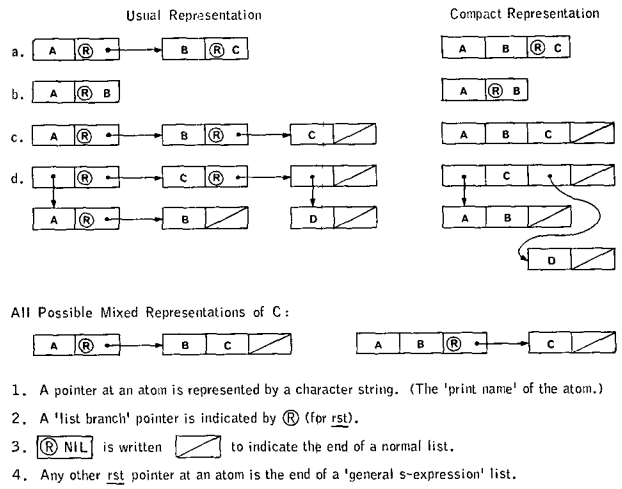

FIG. 1. Comparison of usual and compact list representations

of the compact list representation. In Section 2 the basic operations on such lists are defined.

When storage is exhausted in a plex processing system, a garbage collector is called to find all active storage and make all inactive storage available for use. The Swym garbage collector converts ordinary lists to compact lists whenever it can. In this process, all active storage is rearranged and relocated at the low end of the plex free storage area. Two versions of the central garbage collection routine are presented in Sections 3 and 4. The first illustrates the basic principles; the second includes all details.

An earlier system permitting compact lists intermixed with chained lists has been reported by N. Wiseman [12]. This system provides for the creation of compact lists, but the garbage collector does not rearrange storage to remove **rst** pointers. Unlike Swym, variables may point at **rst** pointers and there may be more than one **rst** pointer between element pointers. But the user must program extra checking to avoid treating **rst** pointers as list pointers. Wiseman presents no data on the effectiveness of his system.

Finally, in Sections 5 and 6 an outline is given of the major techniques used in implementing Swym. Macro instructions made the system easier to write, easier to modify, and easier to read than if it were written entirely in assembly language. Several classes of the Swym implementation macros are described in Section 5. In Section 6 there is a discussion of the major principles which must be considered in the design of a plex processing system implementation.

1. Swym Structures

To illustrate Swym capabilities, an interpreter was implemented for a subset of LISP 1.5 [5]. Very complex plexes can be realized under Swym, but in this section we consider only those required for the interpreter: lists and atoms. A *list* is a sequence of pointers. Each pointer is the address of an element of the list. An element, in turn, can be either a list or an atom. An *atom* is a plex with arbitrary internal structure. Note that lists are specially distinguished plex structures because the garbage collector can compact them.

Swym list words have the format shown in Figure 2a. If the **rst** bit is zero, the word points at an element of the list. If the **rst** bit is one, this pointer is a so-called "list branch" pointer; it points not at an element of the list but at the continuation of the list. The atom bit is one in a pointer at an atom. This is the distinguishing characteristic of an atom in the Swym system. If both the atom and **rst** bits are zero, the pointer points at a sublist of the given list. If both the atom and **rst** bits are one, the end of the list has been reached. A list ending with a pointer at the atom NIL is a normal list; otherwise, it is what in LISP 1.5 is sometimes called a general s-expression. The atom NIL is treated as a list with no elements.

Associated with each atom is an atomhead—a word containing the type of the atom and two marking bits for the garbage collector. Two atom types have been implemented: LISP 1.5 atoms (called symbols) and bit-string atoms. There are three subtypes of the latter: character strings, numbers, and hexadecimal numbers. Other atom types may be defined simply by coding the primitives to create, manipulate, and garbage collect each new atom type. The format of an atomhead is shown in Figure 2b. The twenty-two unused bits may be used for different

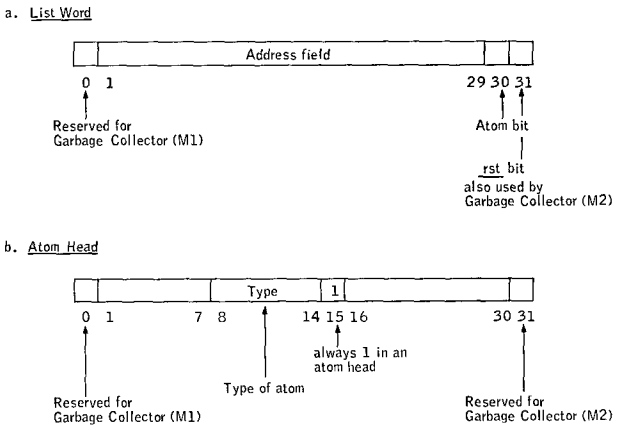

FIG. 2. Formats of Swym internal encodings

purposes for different atom types. For bit-string atoms, they contain the subtype and the length of the string. For symbols, they contain information about the content of the property list.

Because free storage is one contiguous block, new plex structure is created from one end of that block. This storage allocation scheme has proved advantageous in the Cogent system [9]. Lists can be created in compact form if all their elements are known. Atoms of any size can easily be created; for example, bit-string atoms are always stored in consecutive bytes. Note that the garbage collector requires only two bits in the atomhead; all other words in an atom structure may be full words.

Swym was originally designed for a 360/67 with 32-bit addresses; but three nonaddress bits are available in every pointer. The low-order two bits are available because they are always zero in a pointer at a full word. The high-order bit is available because negative addresses are impractical —they are algebraically smaller, though topologically larger, than positive addresses. This discontinuity requires extra code in all address computations if the program might be loaded across the address boundary.

A pointer at an atom points six bytes in front of the atomhead. This assures that the atom bit is one in a pointer at an atom. At the same time, operators on atoms can address the atom directly without masking the atom bit. From a paged-memory standpoint, the atom bit has a small advantage: whether or not an element is an atom can be decided without accessing that element. A major advantage of the atom bit is that **fst** and **rst** both cause an interrupt if their argument is an atom. The advantages of the atom bit suggest its use even in a 24-bit address machine.

2. Basic Operators

The basic Swym operators are direct counterparts of the basic LISP operators, though the mnemonics were changed as an advance beyond the IBM 704. Each operator exists in two forms: first, as a routine which can be called from an interpreted function; second, as a macro which can be coded in assembly language or compiled by a compiler. This second form is the main consideration below; only operators required to describe the garbage collector are presented. A simple interpreter has been implemented, providing the capabilities of basic LISP (without PROG).

fst (l)—returns the **first** element of the list l (corresponds to LISP 1.5 CAR). Processing is interrupted if l is an atom.

rst (l)—returns the **rest** of the list l beyond the first element (corresponds to LISP 1.5 CDR). The program is interrupted if l is an atom.

cons (e l)—**cons**tructs a new list by adding the element e in front of the list l. Returns **a** pointer to a new list $l1$ such that **fst** ($l1$) is e and **rst** ($l1$) is l. If free storage is exhausted, **cons** calls the garbage collector. (**cons** takes two words from free storage: a pointer at e and a **rst** pointer at l.)

rplf (l, e)—**repl**aces the **f**irst element of l with e (corresponds to LISP 1.5 RPLACA). This is accomplished by changing the list l (by storing a pointer to e in place of the pointer to **fst**(l)). The result returned by **rplf** is l, but **fst**(l) is different. **rst**(l) is unchanged.

atom (x)—This predicate tests whether x is an **atom**. To the interpreter, the result is either the atom T or the atom NIL, for true or false, respectively. For the result of a predicate macro, see Section 5.

null (x)—This predicate tests whether its argument is the atom NIL, returns results in the same manner as does **atom**.

Most of these operators are implemented in a straightforward manner. The operators **fst, rplf, atom,** and **null** require 1, 1, 3, and 2 instructions, respectively. **cons** is always a subroutine call because it is lengthy and infrequently used by the system. A special subroutine linkage requiring five instruction-executions calls **cons**. The routine itself requires six instruction-executions unless the garbage collector is called.

Unlike any other routine, **rst** invokes out-of-line code with a simple BAL (Branch And Link). This code is in a section of the system that is always addressable. A call on **rst** is:

```
        BAL     L, RST
```

At this point, the pointer whose **rst** is to be taken must be in register X. At location RST, the code is:

```
RST     TM      7(X),X'01'      test rst bit
        BO      RST1            branch if one
        BXH     X,C4,0(L)       incr X by 4 and return
RST1    L       X,4(X)          pick up rst pointer
        BCTR    X,L             turn off rst bit and return
```

Register L is a standard linkage register. Register C4 is an odd register containing 4. The result of **rst** is left in register X. Either the BXH has added 4, so that X points at the next location of memory, or the L has loaded the next location of memory. Note that in the latter case the BCTR turns off the **rst** bit. If X points to an atom, **rst** will cause a program interrupt: the L is attempted because the TM detects the "1" in the middle of the atomhead. This is why an atom pointer points six, not two, bytes in front of the atomhead. To avoid register transfers, there is a different **rst** routine for each pointer register.

It is a shortcoming of compact lists that the operation **rplr** is not allowed (this would correspond to the LISP operation RPLACD). This is because there is not always a **rst** field to be replaced. As a result, lists cannot be sorted by changing the links between the elements. One possibility is to chain lists in the **fst** direction and use **rplf** to change the order of the list. But this makes for unnatural list structures and nonintuitive coding. A possible solution is the introduction of the atom type true-list, a list with explicit **rst** pointers. Whenever a true-list is created, it is placed on a special list of all such atoms. The first task of the garbage collector is to mark all elements of all true-lists so they will not be collected by other routines. The marking bit **ml** can be used for this function because COLLECT ignores any structure marked with that bit. Another possible solution to the problem posed by **rplr** is to extend the notion of the **rst** pointer. Every time a pointer is used it is checked to see if it points at a **rst** pointer. If so, the **rst** pointer replaces the pointer in question. But this solution greatly increases all list access time.

On the surface, it might appear that arrays are possible as a direct extension of Swym compact lists. A list guaranteed to be compact could be referenced by direct-address arithmetic. Unfortunately, problems arise: if any substructure of the list includes a pointer to a **rst** of that list, that

rst will be collected before the preceding portion of the list. As with true-lists, a possible solution is the creation of a special type of atom called the list-array, a list with no **rst** pointers. When a list-array is created, it is put on a special list. The first task of the garbage collector is then to mark all list-arrays so they will be collected linearly. Again, the **ml** bit could be used.

3. Basic Garbage Collector Algorithm

The Swym garbage collector creates a set of structures isomorphic to all active structures with respect to **rst** and **fst**. Most **rst** pointers are eliminated. This set of structures is in a new core image, created sequentially and written to a temporary storage device. After collecting all active structures, the new core image is read into one end of the plex storage area. The remainder of that area becomes the free storage area so that plexes of arbitrary size may be allocated as required.

An MIT memorandum by Minsky [6] influenced the design of the Swym garbage collector. He suggested both the use of temporary storage and the possibility of rearranging lists so that the **rst** points to the next location of memory. Without suggesting compact lists, he remarks, "There are probably some important applications of (such rearranged lists)." But Minsky's algorithm does not work for even the simplest cases (for example, the structure of Figure 4). The Swym garbage collector works for not only the simplest cases but also the most complex cases of mutual circularity. The complete garbage collector is described in Section 4; in the present section we give a minimal version of the garbage collector to illustrate the central ideas. This minimal version is satisfactory only for structures that never have more than one pointer at any given word of the structure.

COLLECT (x), the portion of the garbage collector presented both here and in Section 4, has as its argument a pointer at a piece of list structure. It then writes that list structure sequentially to the new core image. Other functions must be written to call COLLECT for each possible pointer at active structure, to collect atoms, and to read in the new core image.

The contents of a list are address pointers to the elements of that list. When a list is written to new core, the contents of that list must be the new core addresses of the elements of that list. Consequently, the elements of a list must be COLLECTed before the list itself can be written to the new core. COLLECT (x) proceeds in two recursively intertwined passes. The first pass applies COLLECT to each element of the list x. The second pass writes the new representation of the list x to the new core image. To remember where a piece of list structure is in new core, its **fst** is replaced (**rplf**) with the address of that structure in new core. The head of an atom is used to store the address of that atom in new core.

```
COLLECT (x) = begin list r, t;  word rstbit := x'00000001';
        r := x;
chkloop: t := fst(r);
        if atom(t) then ATCOL (t) else COLLECT (t);
        t := rst(r);
        if atom(t) then ATCOL (t)
        else begin r := t;  goto chkloop end;
        r := x;
wrloop: t := fst(r);
        rplf (r, if atom (t) then GCPUT (HD (t))
                 else GCPUT(fst(t)));
        t := rst (r);
        if atom (t) then GCPUT(HD(t) ∨ rstbit)
        else begin r := t;  goto wrloop end
end COLLECT
```

FIG. 3. Simplified COLLECT algorithm

Three operators must be defined in order to describe the garbage collector:

ATCOL (x)—x must be an atom. If x has not been garbage collected, it is collected and written to the new core image. The atomhead of x is replaced with the address of x in the new core. ATCOL calls separate routines to garbage collect each type of atom.

GCPUT (x)—x is any full word. This word is written to the next available location in the new core image. The value of GCPUT is the address of that location. An internal variable is advanced to point at the next available location in the new core image. GCPUT handles I/O and writes buffers to the external device when necessary.

HD (x)—x must be an atom. HD returns its atomhead; after ATCOL, the atomhead contains a pointer to x in the new core. If x is nonatomic, processing is interrupted.

The basic garbage collection algorithm is given in Figure 3 in a notation derived from ALGOL and LISP M-expressions. The declarator **list** declares a variable which may point at a piece of list structure. The declarator **word** declares a variable whose value is one full word. Note that **rstbit** is initialized to the value 1. This corresponds to the value of a word with just the **rst** bit on. **rstbit** is used to "or" the **rst** bit into a word written to the new core image. The result of applying COLLECT to a simple structure is shown in Figure 4.

"Garbage collection" is truly a misnomer for this algorithm. COLLECT examines only the active list structures, while the garbage is completely ignored and has no effect on the processing. "Storage reclamation" describes the process no better. Possibly better terms might be "storage reorganization" or "garbage control." But the term "garbage collection" is so widely used and so colorful as to preclude replacement.

4. The Complete Garbage Collector Algorithm

The simple garbage collector above is inadequate for many common list structures: circular lists, several lists with the same **rst**, a structure which is an element of more than one list, and more pathological cases. The implemented garbage collector handles all possible cases with marking bits and a fixup table.

Two marking bits are associated with each list word. Each pass sets a marking bit to indicate it has visited a

given word. The first pass sets bit **m1**, the second sets **m2**. Special action must be taken when a marked word is encountered, because that word is already being processed at some other level of recursion. A word with **m2** set always contains the address of the corresponding word in the new core image.

Several functions set and test the marking bits:

MARK1 (w) — The word pointed at by w is marked with **m1**.

MARK12 (w) — The word pointed at by w is marked with both **m1** and **m2**.

UNMARK1 (w) — **m1** is turned off in the word pointed at by w.

M1 (w) — This predicate is **true** if **m1** is on in the word pointed at by w.

M2 (w) — This predicate is **true** if **m2** is on in the word pointed at by w.

Conceptually, each of these functions tests its argument to see if it points at an atom, and adjusts the addressing appropriately. In practice, it is known a priori whether the argument is an atom, and a bit macro (see Section 5) is coded instead of a function call.

In circular structures, a word points at some structure already being collected at some higher level of recursion (**m1** is set, but not **m2**). That word cannot be written correctly to the new core image because its contents are not determined. In most reasonable applications, the number of such circularities is well below one percent of the number of pointers. Nonetheless, some provision must be made to handle this case; in Swym, the garbage collector uses a fixup table. When the correct new contents cannot be determined, a word of zeros is written to the new core and an entry is made in the fixup table. Each entry is two pointers. The first points at the word of zeros in the new core; the second points at the word in old core which will eventually contain the correct address to substitute for the word of zeros. After COLLECT is finished, the second pointer of each fixup entry is replaced by the contents of the word it points at. Then, after the new core image has been read in, the fixups are applied; i.e. the second word of the entry is "or"ed into the location indicated by the first word of the entry. (The "or"ing permits the word of zeros to have the **rst** bit on if required. The fixup procedure thus works for both **fst** and **rst** fixups.)

One new function must be defined to describe the complete garbage collector:

FIXUP (p,c)—the word c (either zero or **rstbit**) is GCPUT to the new core. An entry is made in the fixup table consisting of the address returned by GCPUT and the pointer p.

The function ATCOL defined in Section 3 must be extended. When ATCOL is entered, the **m1** is set in the atomhead. After collecting the atom, both marking bits are set. Since COLLECT may be called for some substructure of an atom, provision is made for a pointer at an atom with **m1** and not **m2** (a fixup entry is generated).

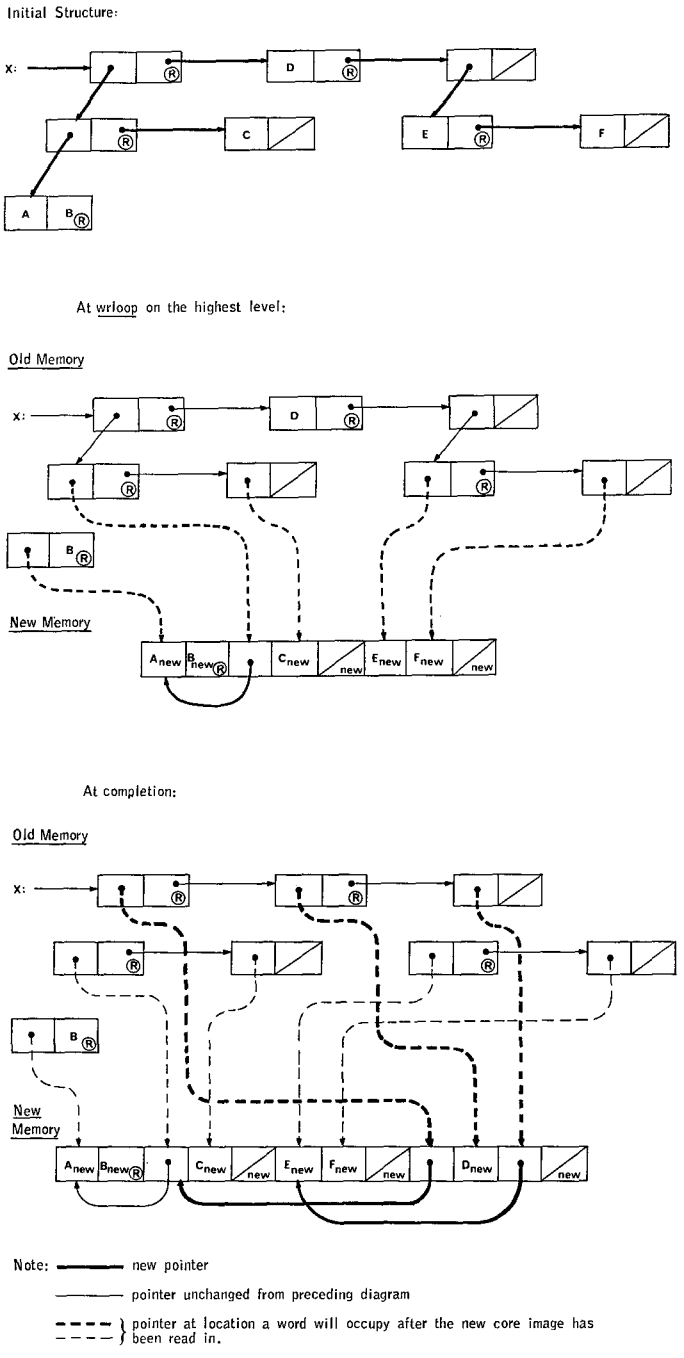

FIG. 4. Effect of COLLECTing a simple structure, assuming all atoms have already been collected. At completion, the new core address of the corresponding structure is in **fst**(x).

The complete garbage collector is given in Figure 5. The argument x must be a pointer at a list structure with neither marking bit on. COLLECT has no value, but the new core address of the list corresponding to x is stored in place of the pointer to **fst**(x). A demonstration that this algorithm creates a correct representation of its argument is given in [2]. The UNMARK1(r) and the Boolean variable m are related. The former indicates the need for a fixup in the **rst** direction; the latter detects this need in the second pass. In Figure 5, the marking bits are assumed

to be associated with each word, and not part of the word. This association could be by extra bits in the hardware or by a bit table in a separate area of memory. The former requires hardware modification, while the latter requires six percent more memory. In the implemented system, the marking bits are in the list words themselves, as shown in Figure 2. Figure 5 must be modified for these bit assignments by turning off the marking bits in the arguments to GCPUT and replacing

t := **rst** (r)

with

if M1(r+4) **then** t := r + 4 **else** t := **rst**(r)

Diagrams of the effect of COLLECT on a complex structure are shown in Figure 6.

Some limited experiments have been conducted with the Swym garbage collector. On one list structure, representing a program, there was a 25 percent saving of storage using compact lists instead of standard lists. This corresponds to an average list length of only two elements. The correspondence is easy to compute. A normal list of length n requires $2n$ pointers. The corresponding compact list requires $n + 1$ pointers, for a saving of $1 - ((n + 1)/2n)$; when n is 10, the saving is 45 percent. Every symbol atom takes at least four words of storage plus the length of the print name; so the number of symbol atoms is also a factor.

For every active word of storage, 40 instructions were executed during garbage collection. This was computed by dividing execution time into amount of active storage. The experimental system did not use external storage; instead, memory was shuttled between two alternate core areas. Thus the time to write out memory is the maximum of the time to write out the active structure and the time to execute 40 instructions for each active word. The time to read in memory is dependent solely on the number of active words.

Several applications for the Swym garbage collector are conceivable, even apart from compact list structure. The

```
COLLECT (x) = begin list r, t; Boolean m;
    word rstbit := x'00000001';
    r := x;
chkloop: comment Loop to collect each fst;
    t := fst(r); MARK1(r);
    if atom(t) then ATCOL(t) else if ¬ M1(t) then COLLECT
      (t);
comment Test for end of list or reached marked word;
    t := rst(r);
    if atom(t) then ATCOL (t)
    else if M2(t) then
    else if M1(t) then UNMARK1 (r)
    else begin r := t; goto chkloop end;
    r := x;
wrloop: comment Loop to write out each new fst;
    m := M1(r); t := fst(r);
    rplf (r, if atom(t) then
                 if M2 (t) then GCPUT (HD(t)) else FIXUP
                   (t, 0)
             else if M2 (t) then GCPUT (fst(t)) else FIXUP
                   (t, 0));
    MARK12(r);
comment Test for end of second pass;
    t := rst(r);
    if atom(t) then
      if M2(t) then GCPUT (HD(t) v rstbit)
      else FIXUP (t, rstbit)
    else if M2(t) then GCPUT (fst(t) v rstbit)
    else if m then begin r := t; goto wrloop end
    else FIXUP(t, rstbit)
end COLLECT
```

FIG. 5. Complete COLLECT algorithm

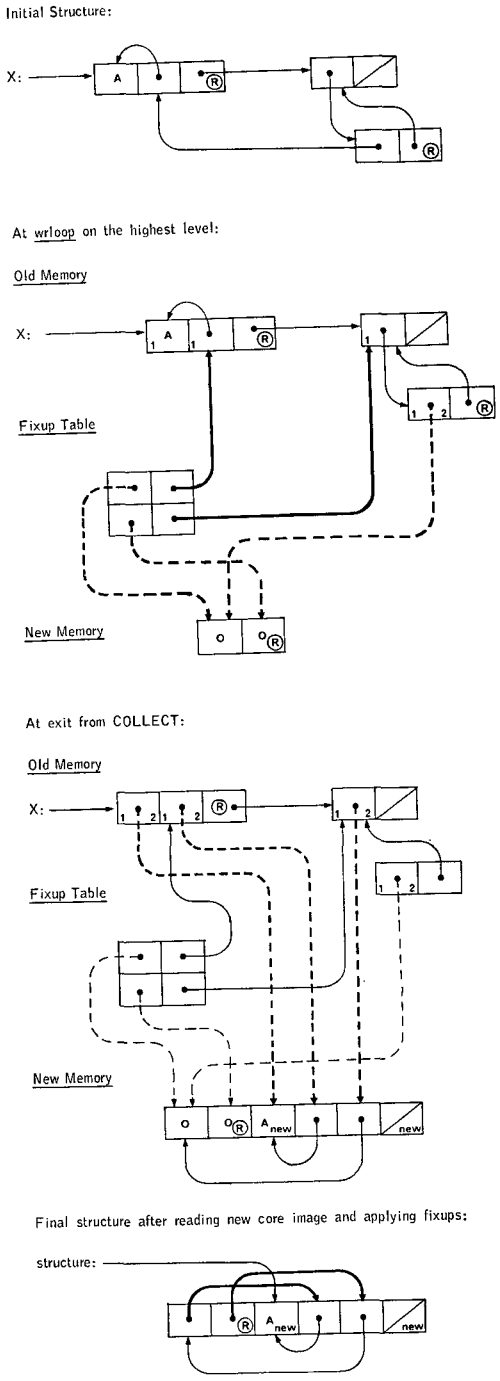

FIG. 6. Effect of COLLECTing a complex structure. Marking bits are indicated by integers in the lower corners. The usual list representation of the initial structure would include a **rst** pointer after the pointer to A.

Swym garbage collector could be valuable in a system with roll out and roll in. If the monitor set a signal for the program to roll itself out, the program could garbage collect for free the next time a **cons** was executed. The garbage collector would be valuable in a paging environment because it tends to localize list structure. External storage of structures has always been a problem for plex processing systems. The Swym garbage collector provides an algorithm for scanning lists and storing them in a compact form on an external device. Another application for this algorithm is in the transmission of list structures between two machines over a slow channel. If the new storage is written starting at location zero, the address fields can be small. Only as the size of the structure passes a power of two would the length of each address field have to increase.

The implemented garbage collector stores partially collected structures on the stack but uses a trick to avoid saving return addresses during recursion. It would be possible to use the WISP technique [10] to avoid using the stack during collection. This was not done because it would involve at least two more passes over the data. In a memory-sharing environment, it is sometimes possible to acquire temporarily the needed extra storage for a stack; otherwise, sufficient stack must be available to hold twice the length of the longest **fst** chain.

5. Macros for Implementing Swym

Macros are a powerful programming tool for more important reasons than just the reduction of the amount of writing required. Operations can be stated in terms of the problem at hand rather than the machine at hand. Even though the macro form of many of the operators in Section 2 assembles only one instruction, it is important to use the macro because the intent of the operation is clarified and the data type of the operand is emphasized. Clarity is important both to the interested reader and the perplexed debugger. Consistent and extensive use of macros also facilitates program modification. In many cases, an entire data representation can be modified by changing a few macros.

Several classes of macros were created to facilitate the implementation of Swym. One class has already been described—the macros implementing the primitive functions. Another class of macros manipulates data on the pushdown list: PUSH, POP, TOP, RPLTOP, etc. To avoid excessive subroutine calls, GCPUT and FIXUP were also implemented as macros. The bit macros and the conditional execution macros deserve detailed discussion. These macros show the power of the 360 assembler, especially the global array facility. The creation of the initial contents of the plex storage area also demonstrated the power of the 360 macros. This process involved hash coding identifiers (at assembly time) to assemble list structure putting those identifiers in the proper bucket of the OBLIST, the list of all atoms.

In the 360 assembler language, a mnemonic may easily represent a specific bit within a byte. But there is no convenient way to specify a bit within a field and to then code instructions in terms of that specification. The Swym bit macros provide this facility—the macro BIT declares a mnemonic to represent a bit in a field. The macros SETBIT, RESETBIT, INVRTBIT, and TESTBIT accept as arguments the name of a register pointing at a field in memory and the name of a bit. The first three affect the state of the bit; TESTBIT is a predicate macro—it tests the bit's state.

The byte-immediate instructions are used to implement these macros. One instruction is assembled for each macro (two for TESTBIT). The BIT definition is preserved in a pair of assembly-time global arrays.

The garbage collection algorithm frequently makes a test and then executes one of two alternate pieces of code. In higher-level languages, such constructions are usually expressed as conditional statements in the form:

if ⟨predicate⟩ **then** ⟨true part⟩ **else** ⟨false part⟩

The advantage of this kind of language facility is that it makes clear that one of two alternate paths will be executed. In conjunction with proper indentation, this facility contributes greatly to the reader's comprehension of the program. To provide this convenient high-level language structure in assembly language, a set of macros was implemented for Swym.

Four macros are basic to the conditional execution facility: IF, THEN, ELSE, and ENDIF. None has any parameters. They are used in an assembly language just as are their counterparts in a higher-level language program:

```
IF
    code for ⟨predicate⟩
THEN
    code for ⟨true part⟩
ELSE
    code for ⟨false part⟩
ENDIF
```

The code for the true and false parts may be any code including other if-then-else sets. But the predicate may not contain IF. It is possible to omit the ELSE and the false part; the correct code will still be generated.

The four conditional macros generate appropriate labels and branch instructions (given the assembler, there is no good way to avoid labels altogether). The labels are generated by the IF macro and saved on a global assembly time stack. The use of the stack permits nesting of IF's up to 63 levels.

More high-level-language-like features are available within the predicate part. The conditions most likely to be tested for are ATOM, NULL, and TESTBIT. Each of these is implemented as a predicate macro. This assembles a test and a conditional branch to one of the labels on top of the IF stack. These macros can also be used without IF by supplying a destination label as an argument. The conditional branching macro PREDBR can be used in case the provided predicates are inadequate. Before calling this macro, the condition code must be set for the condition to be tested. PREDBR's arguments are two op codes, one to make the test for true, the other to make the test for false (e.g. BH and BNH). PREDBR chooses the

test for false unless the NOT macro has occurred. The branch address is taken from the global IF label array and one instruction is assembled. All predicate macros use PREDBR to do their branching. Elementary predicate macros may be combined by the use of the Boolean connective macros AND, OR, and NOT. These macros have no parameters. There are no parentheses macros, and AND has higher precedence than OR.

The macros described in this section simplified construction of the garbage collector and made the result easier to read. The code is written beginning in columns 40, 45, 50, 55, depending on the level of nesting of IF's. This separates the source program from the generated code. Few comments are required because the macros are highly mnemonic and specific to the data structure.

6. Factors in the Design of a System Implementation

While implementing Swym, several factors important to the design of any system implementation were isolated. The more important of these factors can be summed up in two rules:

1. Optimize linkages.
2. Identify pointers.

Optimization effort should be spent on the most frequent operations. In plex processing languages, the most frequent operations are not the callable operations (**fst, rst, cons,** etc.), but the linkage operations: data access, routine linkage, and variable binding. These operations are so prevalent that serious consideration must be given to designing them into the hardware. Each of these is discussed below. The second rule is a result of the fact that the garbage collector must be able to find all active structures. The implications of this rule are also discussed below.

Some data representations require address decoding to access a data element; some even require table look-up. Since any operation on data requires accessing that data, such time-consuming access methods mean that programs cannot run rapidly. This usually means that the program cannot be used to attack any problems big enough to justify the use of a computer. In plex processing, a better phrase than "data access" is "descend one level in the data structure." The Swym operators **rst** and **fst** descend one level in the data structure and are reasonably efficient (**rst** involves at most five instruction-executions). Swym also provides for storing data in atoms; the components of an atom can be accessed directly if the address of the atom is known.

Plex processing programs tend to execute a large number of subroutine calls; so the routine linkage mechanism must be efficient. While it is a waste of both time and memory to save and restore 16 general registers every time a subroutine is called, it is sheer nonsense to call a special linkage subroutine every time a subroutine is called. If any function of a machine is considered when that machine is designed, that function ought to be routine linkage. No more than two instructions should be executed to call a subroutine: one to call the routine, and a second to exit from the routine. Furthermore, these instructions must be satisfactory for recursive routines; that is, return addresses must be saved on a stack. The Swym system has three routine linkage macros: one each for entry, exit, and call. Each assembles three instructions. Altogether, execution of the nine instructions requires accessing only 11½ words of memory. This is probably the minimum required to perform these necessary functions: maintain addressability for the calling and called routines, stack and unstack the return address, and mark the return address to distinguish it from a pointer.

The Swym system supports several different variable-binding methods, but most are beyond the scope of this paper. The system itself binds variables to general registers, locations on the stack, and a special section of memory for a few global values. General registers to be saved must be saved and restored by the calling routine. With this convention, very few registers need be saved because only those registers with active information are saved. Moreover, a register can be saved and more than one function called before that register is restored.

Because all active storage must be COLLECTed, all pointers must be recognizable as pointers. This is a stringent restriction on system design. Some means must be provided to determine for each word in the stack or in a plex whether or not it is a pointer. A word can be known to be a pointer either by testing special bits or by the context in which the word is accessed. In Swym, all list words are pointers because lists can contain only pointers. It is simple to define atom types having pointers known to be pointers because of their position in the plex for that atom type. Bits are used to distinguish pointers on the Swym stack. A low-order bit of one indicates that that word is not a pointer. Return addresses are marked this way; they point one byte in front of the return location. Blocks of words may be allocated on the stack. Bits in the stack-block head indicate which words in the block are pointers.

Conclusion

In this paper we have discussed a new list representation, a garbage collector especially suited to that representation, and some of the considerations involved in designing a system implementation. The data representation has some advantages in saving space. The garbage collector has the same advantage, as well as the advantage of putting associated information close together. These advantages must be weighed carefully against the slight amount of additional processing entailed. The most advantageous use of these structures is in situations where information must be transferred; many instructions can be executed in the time needed to do one word of input/output. Swym demonstrates that careful attention to the overall design of a system implementation can contribute to its success.

Acknowledgment. The author is grateful for the advice and encouragement of Professor William F. Miller.

RECEIVED NOVEMBER, 1968; REVISED JANUARY, 1969

REFERENCES

1. GRAY, J. C. Compound data structure for computer aided design; a survey. Proc. ACM 22nd. Nat. Conf. 1967, Thompson Book Co., Washington, D. C., pp. 355–365.
2. HANSEN, W. J. The impact of storage management on the implementation of plex processing languages. Tech. Rep. No. 113, Computer Science Dep., Stanford U., Stanford, Calif., 1969.
3. KNUTH, D. E. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Menlo Park, Calif., 1968.
4. LANG, C. A., AND GRAY, J. C. ASP—A ring implemented associative structure package. *Comm. ACM 11*, 8 (Aug. 1968), 550–555.
5. MCCARTHY, J., ET AL. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.
6. MINSKY, M. L. A LISP garbage collector using serial secondary storage. MIT Artificial Intelligence Memo. No. 58, MIT, Cambridge, Mass., Oct. 1963.
7. Ross, D. T. A generalized technique for symbol manipulation and numerical calculation. *Comm. ACM 4*, 3 (Mar. 1961), 147–150.
8. ——. The AED free storage package. *Comm. ACM 10*, 8 (Aug. 1967), 481–492.
9. REYNOLDS, J. C. Cogent programming manual. Argonne Nat. Lab. Rep. No. ANL-7022, Argonne, Illinois, Mar. 1965.
10. 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.
11. STYGAR, P. LISP 2 garbage collector specifications. TM-3417/500/00, System Development Corp., Santa Monica, Calif., Apr. 1967.
12. WISEMAN, N. E. A simple list processing package for the PDP-7. In *DECUS Second European Seminar*, Aachen, Germany, Oct. 1966, pp. 37–42.

A Base for a Mobile Programming System

RICHARD J. ORGASS
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

AND

WILLIAM M. WAITE
University of Colorado, Boulder, Colorado

An algorithm for a macro processor which has been used as the base of an implementation, by bootstrapping, of processors for programming languages is described. This algorithm can be easily implemented on contemporary computing machines. Experience with programming languages whose implementation is based on this algorithm indicates that such a language can be transferred to a new machine in less than one man-week without using the old machine.

KEY WORDS AND PHRASES: bootstrapping, macro processing, machine independence, programming languages, implementation techniques
CR CATEGORIES: 4.12, 4.22

1. Introduction

The development of many special purpose programming languages has increased the overhead associated with changing computing machines and with transferring a programming language from one computer center to another. A number of writers have proposed that these languages should be implemented by bootstrapping. (Since this work is well known, it is not summarized in this introduction.)

The description is given of a bootstrapping procedure which does not require a running processor on another machine for its implementation. A compiler is described which can be trivially implemented "by hand" on contemporary computing machines. This simple compiler is then used to translate a more elaborate one, and so forth, until the desired level of complexity is reached.

This paper is a complete description of the first stage of such a bootstrapping procedure. It is organized as follows: in Section 2, the processing performed by the simple compiler (SIMCMP); in Section 3, some examples of its use; in Section 4, the environment which must be coded for a particular computing machine; and in Section 5, the SIMCMP algorithm.

2. Specifications for SIMCMP

The algorithm described in this paper was constructed to provide a compiler of minimum length and complexity which is adequate to serve as the base for the implementation, by bootstrapping, of programming systems. SIMCMP is essentially a very simple macro processor. Source language statements are defined in terms of their object code translations. These definitions are stored in a table. A source program is translated one line at a time. The input line is compared with all of the entries in the table. When a match occurs, the object code translation of the input line, with appropriate parameter substitutions, is punched. SIMCMP can be used to translate any source language which can be defined by means of simple substitution macros with single character parameters.

Several terms as they are used in this paper are illustrated here. In a conventional macro processor, a *macro definition* is of the form:

MACRO NAME (P_1, $\cdots$, P_n)
Code body
END

The strings 'MACRO' and 'END' serve to delimit the macro definition. 'NAME' stands for the *name of the macro*, and 'P_1', $\cdots$, 'P_n' stands for the *formal parameters* of the macro. This macro is *called* by a line of the form 'NAME (A_1, $\cdots$, A_m)' where m is less than or equal to n. When this *call* is encountered in the program text, the code body of the macro NAME is *evaluated* by sub-