UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP None

3. REPORT TITLE

A New List-Tracing Algorithm

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Technical Memorandum

5. AUTHOR(S) *(Last name, first name, initial)*

Fenichel, Robert R.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
October 1970	26	13

8a. CONTRACT OR GRANT NO. Nonr-4102(01) b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) MAC TM-19 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

List-processing systems have each allowed use of only a single size and configuration of list cell. This paper describes a system which allows use of arbitrarily many different sizes and configurations of list cell, possibly not specified until run time.

14. KEY WORDS

List-Processing   Garbage Collection   Storage Allocation

DD FORM 1 NOV 65 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

ACKNOWLEDGMENT

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

A NEW LIST-TRACING ALGORITHM

MAC Technical Memorandum 19

Robert R. Fenichel

October 1970

PROJECT MAC

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

ABSTRACT

List-processing systems have each allowed use of only a single size and configuration of list cell. This paper describes a system which allows use of arbitrarily many different sizes and configurations of list cell, possibly not specified until run time.

1. Introduction

List-processing systems (*e.g.*, LISP 1.5 [10], SLIP [12]) have each allowed use of only a single size and configuration of list cell. (1) This paper describes a system which allows use of arbitrarily many different sizes and configurations of list cells, possibly not specified until run time.

Multiple sizes and configurations of list cells are important in many applications where the natural quanta of data are not homogeneous in size and format. For example, an algebraic interpreter might record the following information about each variable known to it:

reference count

value

print name

pointer to hash-table entry

The interpreter might also handle floating-point numbers as objects. Now, if only a single cell-size is allowed, then either floating-point numbers will be represented with extravagant waste of space, or variables will each be chained out into several

---

(1) The aborted LISP 2 system [1] was to have allowed multiple cell sizes and configurations, but only with a bit of "systems programming ... beyond the domain of the average user" [5, p. 7] supporting each type of cell.

smaller cells, with concomitant waste of both space (for spurious pointers linking the cells) and time (for following the spurious pointers).

It will appear (Section 2 below) that list-tracing is the primary obstacle to implementation of list-processing systems with multiple sizes and configurations of cells. Sections 3 and 4 describe a technique for list-tracing in such systems. (1)

The remainder of the paper is generally tutorial, relating the list-tracer of the earlier sections to existing problems and solutions in list-processing.

2. The Importance of List Tracing

Any list-processing system must provide means for obtaining single list cells from free storage (<u>nucell</u>), for setting and examining the contents of the various fields of a given list cell (<u>set</u>/<u>look</u>), and for returning disused list structure to free storage (<u>eraselist</u>). Other services are generally defined in terms of these primitives. (2)

---

(1) After developing this technique, the author became aware of the somewhat related work of S. Marshall [8].

(2) Here and below, services not related to list processing are ignored. For example, only a small portion of LISP is relevant; most of LISP is concerned with function-application, arithmetic, and other extraneous services.

2.1 Nucell

Obtaining cells from free storage is a relatively well-understood problem, even when cells of unpredictable size must be delivered. [6, p. 435-451] In each traditional list-processing system, this service is provided by a built-in function; (1) in modern, general-purpose programming languages, run-time routines are necessarily provided to perform this service. (2)

2.2 Set/look

Setting and examining the contents of fields within list cells has never been even a code-optimization problem. In each traditional list-processing system, there are a setting function and an examining function for each field of the standard list cell. (3) Adding a new type of cell to such a system entails adding a few more setting and examining functions to the library.

---

(1) E.g., cons in LISP and nucell in SLIP.

(2) E.g., The routines supporting ALLOCATE statements in PL/1 [2] or the FREE (sic) routine in AED [11].

(3) In LISP, for example, the number of fields is two, so the system provides two examining functions (car and cdr) and two setting functions (rplaca and rplacd).

In systems embedded within modern general-purpose languages, these numerous built-in functions are not necessary. Using based-structure declarations(1) within such languages, fields of arbitrary list cells may be referenced by mnemonic names, and accessed by in-line code.

2.3 Eraselist

Returning disused list-structure to free storage must be performed by some variation of one (or both [13]) of the following two methods:

2.3.1 Garbage collection [9]

When free storage is in short supply (or, as in [4], when it is undesirably scattered), trace all list-structure accessible by program, marking all cells touched. Then scan the entire region of memory from which cells are taken. During this scan, place unmarked (= inaccessible) cells on the free-storage list, and remove the marks from marked cells.

---

(1) This terminology is that of PL/1. Users of AED or of SAL [7] declare <u>components.</u>

2.3.2 Discard-Scanning [12]

When a piece of list-structure is discarded by a user program, trace through this structure and return all of the cells involved in it to free storage. As a refinement (as in [12]), allow cells to hold reference counts, and return substructures to free storage only when they are no longer shared as substructure by non-discarded structures.

Either of these methods requires a procedure for tracing through a list structure. (1) Such a procedure must be able to accept a pointer to a cell _C_ and, using this pointer, to explore _C_, identifying all of the other cells which have _C_ as list-parent. The procedure must not be misled by the presence in _C_ of irrelevant fields containing floating-point numbers, flag-bits, or other non-addresses.

The list-tracing procedure is the only essential list-processing service which is not routinely provided by

---

(1) The original SLIP implementation [12] spreads the trace out thinly and discontinguously in time. This is in pleasing accordance with the engineering principle which favors smooth, continual application of energy. The total effort is the same, however, and the information needed by the procedure is unchanged.

modern general-purpose languages.

3. Pointers and Type Information

Even though each traditional list-processing system allows only a single <u>size</u> of cell, none gets by with only a single <u>type</u> of cell. In LISP, for example, there are atoms and non-atoms. In SLIP, there are headers and non-headers.

Given a pointer to a cell <u>C</u>, the type of <u>C</u> must be computable. In early implementations, this necessary type information was recorded in <u>C</u> itself.

In more recent implementations, the type information is carried in the pointer to <u>C</u>. This use of <u>rich pointers</u> reflects two developments in list-processing implementation [3]:

3.1 Immediate values

If an atomic datum can be expressed in as few bits as an address,(1) then it is more efficient to copy such a datum than to handle it indirectly. But if "pointers" sometimes contain immediate data instead of addresses, then

(1) Truth values and small-magnitude integers exemplify such data.

pointers must also contain type information to characterize the data they hold.

3.2 Virtual memories

Querying type-information is a common list-processing operation. (1) In virtual-memory systems, where memory references may be costly, it is economical to place type-information in the pointers, so that references to the cells need not be made.

4. <u>An Elementary List-Tracing Procedure</u>

4.1 Definitions

4.1.1 A <u>word</u> is a quantity of memory sufficient to hold an address.

4.1.2 A <u>pointer</u> is a two-word object consisting of a type-code and an address.

4.1.3 A <u>cell</u> is a set of one or more contiguous words in memory. The requirement of contiguity is imposed only so that a single address somehow specifies the whole cell.

---

(1) <u>E.g.</u>, the functions <u>atom</u>, <u>numberp</u>, <u>fixp</u>, etc. in LISP, or <u>namtst</u> in SLIP.

4.1.4 A *word in a cell C* may be used

(a) together with the next word in *C,* to contain a pointer, or

(b) to contain the address of another cell *D,* or

(c) to contain bits (*e.g.,* a floating-point number) which are neither the type-code of a pointer (as in (a)) nor the address of another cell *D* (as in (b)).

4.1.5 Two cells *C* and *C'* are of the same *type* if and only if

(a) they are of equal size and

(b) for each word *W* in *C,* with corresponding word *W'* in *C'*,

(i) if *W,* taken together with the next word in *C,* contains a pointer, then so do the corresponding words in *C'*.

(ii) if *W* contains the address of another cell *D,* then *W'* contains the address of a cell *D'* which is of the same type as *D.*

(iii) if *W* contains bits which are neither the type-code of a pointer nor the address of another cell _D_, then so does _W'_.

4.1.6 Two pointers point to cells of the same type if and only if they contain the same _type-code._

4.2 Templates

Cell-type _T_ can be described with a _template_ of _n_+1 words, where _n_ is the number of words in each cell of type _T._

(a) Word 0 contains _n._

(b) For $1 \leq i \leq n$,

(i) If words _i_ and _i_+1 of each cell of type _T_ contain a pointer, then word _i_ of the template contains _P_, a constant different from any type-code.

(ii) If word _i_ of each cell of type _T_ contains the address of a cell of type _T'_, then word _i_ of the template contains the type-code of _T'_.

(iii) If word _i_ of any cell of type _T_ contains bits which are neither the type-code of a pointer nor the address of another cell _D_, then word $\underline{i}$ of the template contains $\underline{Z}$, a constant different from $\underline{P}$ and from any type-code.

4.3 The Key Idea

In order to allow a list-tracing program to trace a list structure containing cells of type $\underline{I}$, the program must, from the pointer to a cell of type $\underline{I}$, be able to find the template for cells of type $\underline{I}$. The simple, single, central idea of this paper is

The type-code for cells of type $\underline{I}$ may be the address of the template for cells of type $\underline{I}$.

4.4 The Algorithm

A list-tracing procedure is displayed here in a bastard form of PL/1. As given, this procedure retraces shared substructures, loops indefinitely on reentrant structures, and is highly recursive.

```
list_scan: procedure (type_code, cell_address, f);
     declare (type_code, cell_address) address,
          f external entry(address, address);

     /* Apply the function 'f' to every cell of the list
     structure whose root is a cell of type 'type_code' at
     location 'cell_address'. */

     declare cell_word(n) address based (cell_address),
```

```
      1 template based (type_code),
            2 n fixed,
            2 template_word(n) address,

      i local fixed,

      (P, Z)  external address;

i=1;
do while (i ≤ n);
      if template_word(i) = Z then
            /* Corresponding word in the cell is a
            bit-pattern which does not specify a
            list-child of the cell. In other words, the
            corresponding word of the cell is irrelevant
            to list-tracing. */

            i = i+1 /* Skip past the irrelevant word */;

      else if template_word(i) = P then
            /* Corresponding word in the cell is a
            type-code, and the following word in the cell
            is an address. Together, these specify a
            list-child of the cell. */

            do;
                  call list_scan(cell_word(i),
                        cell_word(i+1), f);
                  i = i+2;
            end;

      else
            /* Corresponding word in the cell is the
            address of a list-child cell whose type is
            given by this word in the template.*/

            do;
                  call list_scan(template_word(i),
                        cell_word(i), f);
                  i = i+1;
            end;
end;

call f(type_code, cell_address);
return;
```

```
end list_scan;
```

5. Some Elaborations of the Elementary Procedure

5.1 Immediate Data (cf. Section 3.1 above)

As shown in Section 4.4 above, the elementary procedure almost allows templates with $\underline{n} = 0$. The only necessary change is to make the call of $\underline{f}$ conditional upon $\underline{n} > 0$.

For added efficiency, of course, the loop could be elaborated so as to avoid spurious self-calls for processing of descendant pointers which contain immediate data in the place of addresses.

5.2 Classes of Types

Certain operations at higher levels of the system may involve classes of types. For example, numbers, arrays, and variables are all "atoms" to LISP. It may be useful to add a word of flag bits to each template so that classes of types may be easily distinguished.

5.3 Re-entrant Lists and Shared Substructure

The elementary procedure will exert redundant effort on shared substructure, and it will exert unbounded effort on reentrant lists. Sometimes these cases may be avoided by using <u>simplified</u> <u>templates;</u> at other times, <u>marking</u> will be necessary.

5.3.1 Simplified Templates

By the definition of <u>reentrant</u> implicit in Section 4.1 above, many structures are reentrant even though they would not ordinarily be so described. For example, consider any structure in which pointers are matched by back-pointers, as in SLIP. Such a structure is its own list-grandchild, and reentrant for purposes of the elementary procedure.

This superficial reentrancy is not noticed in traditional systems since hand-tailored list-tracers have always been sensible enough to follow only one set of pointers. To establish this restriction here, the back-pointer words of the appropriate templates can be set to <u>Z</u> ("this is not an address"), even though the corresponding words in the cells do really contain addresses.

5.3.2 Marking

When true reentrancy is possible, the list-tracing procedure must <u>mark</u> traced lists so that they are not traced again as their own list-descendants. Marking may also be desirable to avoid redundant tracing of shared substructure.

5.3.2.1 It may happen that cells of certain types are never the roots of reentrant or shared structures. Cells of these types need never be marked.

5.3.2.2 If the list-tracer is making use of marking, it will interrogate a bit in the template to see if this cell should be marked.

5.3.2.3 If cells of this type should be marked, then the location of the mark must be determined. The location may be set by convention (e.g., any cell which is marked is marked in its first word) or by the presence of a special code (like P and Z) in the corresponding word of the template.

If this cell is already marked, then the list-tracer returns. Otherwise, the list-tracer marks this cell and traces its substructure.

Each use of a marking list-tracer may need to be followed by a list-trace to reset the marks. Alternately, if a broad field is used for marking, then each trace can use a new bit-pattern (say, consecutive integers) as the mark. (1)

5.3.3 Reference Counts

The use of reference counts for storage management [12] is formally similar to the use of marking. In particular, the considerations of paragraphs 5.3.2.1-3 are all applicable to reference counts.

5.4 Linear Lists

Linear lists are common in list-processing applications. In other words, it is common for each cell of a given type to utilize a given word for the address of another cell of the same type. To indicate the end of the list, a distinctive bit-pattern (e.g., all zero) is used.

It may be economical to use a bit in the template to indicate that the associated cells are members of linear lists. Then, the list-tracer can use a high-speed loop instead of a self-call for each cell of linear list.

(1) This technique is due to Martin Richards.

5.5 Other Special Cases

In certain applications, the list-tracing program may need to take account of idiosyncratic properties of certain cell-types. For example, suppose that a list-scanner is being used to scan discarded structures (Section 2.3.2 above) in an algebraic interpreter which includes cells of type <u>variable.</u> One of these cells should be returned to free storage only if

(a) its reference count has gone to zero, and

(b) there is no value associated with this variable.

Some variables may additionally be specially protected from disappearance.

The interpreter may have all of the variable cells chained to an identifier hash table. If this is so, then when a variable cell is to be returned to free storage, the hash-table chain must be patched.

The various actions of this example might all be triggered by a bit in the template of cells of type <u>variable.</u> Diffidence about giving this sort of application-dependent information to the list-processing routines may be offset by the arguments of Section 7 below.

6. <u>Non-Recursive List Scanning</u>

In some environments, it will be economical to rewrite the list-scanner so as to avoid recursive calls.

6.1 Local Stacking

If memory usable as a <u>stack</u> is available, each recursive call of the list-scanner can be replaced with a "push" operation of the pointer argument. Then the list-scanner code is all surrounded by a loop which takes pointers from the stack until the stack is empty.

6.2 The Deutsch-Schorr-Waite Algorithm

If a stack is not available, the list-scanning technique of Deutsch, Schorr, and Waite(1) may be appropriate. The Deutsch-Schorr-Waite (DSW) technique uses the scanned list itself for temporary storage during the scan.

6.2.1 The Algorithm

Consider a list structure consisting (in part) of a grandparent cell <u>G,</u> a parent cell <u>P,</u> and a child cell <u>C.</u> To scan cell <u>C,</u> the list-scanner needs

(1) It is credited to them in [6], p. 417.

(a) the type of C, and

(b) the address of C, and

(c) an index which will range over the words of C.

The DSW list scanner also remembers these data for cell P.

When the DSW list-scanner returns to P from C, it must

(a) discard its data concerning C, and

(b) replace its data concerning C by its data concerning P, and

(c) somehow replace its data concerning P by the corresponding data concerning G.

These data concerning G are retrieved from a word of cell P, where they were stored when the DSW scanner was about to move from P down to C. The word used for these data is simply that word of P containing the address of C. Since the data concerning C are kept alive within the DSW program during operations on C, continuous storage of these data in P is not essential. Of course, these data must be restored to P upon return from C, before they are "discarded" as suggested just above.

6.2.3 Applicability of the Algorithm

The DSW scheme may require that a spare word or two be present in every cell. This is so because although three data (type, address, and internal index) concerning the grandparent cell must be temporarily stored in the parent cell, only one datum (address) concerning the child cell may be there to be overwritten. At best, the child cell will have been identified with two words (type and address), and the grandparent's internal index will still be homeless.

In practical cases, there are frequently a few extra bits available in each cell. It might be noted, moreover, that the internal index may be representable with as little as one bit; this was its size in the original DSW implementation.

7. <u>On Subroutines and Technical Communication</u>

Most of the various list-processing techniques described above have been implemented for use in an algebraic interpreter. The interpreter uses cells of seven different sizes and about fifty different types. This paper, instead of being wholly devoted to discussion of the underlying techniques, might have more concerned itself with the modularization, calling-sequences, and other black-box details.

This technique-oriented description is the result of a view about the state of programming. A few years ago, list-processing was an arcane and difficult business. Users were not interested in the internal techniques, any more than, a few more years ago, they had been interested in the internal techniques of floating-point interpretive routines.

Today, the techniques once used only within specialists' floating-point routines are coded in-line by everyone. Similarly, list-processing code is today more often tailored than bought off the rack.

The listing-lovers, no doubt, would be in touch with the author no matter what he did.

8. Acknowledgements

The author is indebted to Prof. J. Moses for his encouragement, to C. Hewitt and G. Sussman for their comments on an early draft, and to Prof. Arthur Evans, Jr., for bringing the work of S. Marshall [8] and Martin Richards to the author's attention.

REFERENCES

1. Abrahams, Paul W. *et al.* "The LISP 2 Programming Language and System," *AFIPS Proceedings,* XXIX (Fall, 1966), pp. 661-676.

2. Anonymous. *PL/1 Reference Manual* (IBM Corporation, 1968).

3. Bobrow, Daniel G., and Murphy, Daniel L. "Structure of a LISP System Using Two-Level Storage," *CACM,* X, 3, (March, 1967), pp. 155-159.

4. Fenichel, Robert R., and Yochelson, Jerome C. "A LISP Garbage Collector for Virtual-Memory Computer Systems," *CACM,* XII, 11 (November, 1969), pp. 611-612.

5. Hawkinson, L. "LISP 2 Internal Storage Conventions." System Development Corporation Tech Memo TM-3417/550/00 (Santa Monica, 1967).

6. Knuth, Donald E. *Fundamental Algorithms* (Reading, Massachusetts: Addison-Wesley, 1968).

7. Lang, Charles A. "SAL-Systems Assembly Language," *AFIPS Proceedings,* XXXIV (Spring, 1969), pp. 543-557.

8. Marshall, S. "An Algol 68 Garbage Collector." Kiewit Computation Center Technical Memorandum TM011 (Hanover, New Hampshire: Dartmouth College, 1969).

9. McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computation by Machine - I," CACM, III, 4 (April, 1960), pp. 184-195.

10. ______ et al. LISP 1.5 Programmer's Manual (Cambridge: MIT)Press, 1965).

11. Ross, D.T. AED-0 Programming Manual (Hectographed, 1964).

12. Weizenbaum, J. "Symmetric List Processor," CACM, VI, 9 (September, 1963), pp. 524-544.

13. ______. "Recovery of Reentrant List Structures in SLIP," CACM, XII, 7 (July, 1969), pp. 370-372.

Richards to the author's attention.