A Method for Overlapping and Erasure of Lists

GEORGE E. COLLINS, *IBM Corp., Yorktown Heights, N. Y.*

Abstract. An important property of the Newell-Shaw-Simon scheme for computer storage of lists is that data having multiple occurrences need not be stored at more than one place in the computer. That is, lists may be "overlapped." Unfortunately, overlapping poses a problem for subsequent erasure. Given a list that is no longer needed, it is desired to erase just those parts that do not overlap other lists. In LISP, McCarthy employs an elegant but inefficient solution to the problem. The present paper describes a general method which enables efficient erasure. The method employs interspersed reference counts to describe the extent of the overlapping.

Introduction

The wide applicability of list processing is just beginning to be fully realized. This utility and versatility is due in large part to the ingenious scheme of representing lists in a computer storage which was devised by Newell, Shaw and Simon (see [4], for example). The primary merit of this scheme is that allocation of storage space for data is synthesized with the actual generation of the data. This is a practical necessity in many applications for which the quantities of data associated with some variables is highly unpredictable. This scheme achieves a sort of local optimization in that each "basic item" of data occupies a minimal amount of space.

A secondary, but often extremely important, merit of the scheme is that data having multiple occurrences often need not be stored more than one place in the computer. One may visualize this situation as the overlapping or intersection of lists, and its utilization may be regarded as a step toward global optimization of storage allocation.

However, the overlapping of lists leads to difficulties and complications in the erasure of lists (that is, in the return of words to the list of available storage). Given the location of a list that is no longer needed, it is desired to erase just those parts that do not overlap other lists. There is no general method of doing this short of making a survey of all lists in memory.

Two methods have been described so far in the literature for the solution of this difficulty, each of which has certain disadvantages or limitations. As a result we have been motivated to devise a new method, described in this paper.

The first method, due to McCarthy, is described in [3]. Briefly, in this method individual lists are "abandoned" (disconnected from a base register) rather than erased. Then, when the list of available storage has been exhausted, a survey is made of all registers currently employed in non-abandoned lists. The complement of this set of accessible registers is then returned to the list of available storage.

McCarthy's solution is very elegant, but unfortunately it contains two sources of inefficiency. First and most important, the time required to carry out this reclamation process is nearly independent of the number of registers reclaimed.[1] Its efficiency thereby diminishes drastically as the memory approaches full utilization.[2] Second, the method as used by McCarthy required that a bit (McCarthy uses the sign bit) be reserved in each word for tagging accessible registers during the survey of accessibility. If, as in our own current application of list processing, much of the atomic data consists of signed integers or floating-point numbers,[3] this results in awkwardness and further loss of efficiency.[4]

The second method, described by Gelernter et al. [2], consists of adopting the convention that each list component is "owned" by exactly one list, and "borrowed" by all other lists in which it appears. A "priority bit" is then used in each "location word" to indicate whether the list entry referenced is owned or borrowed. An erasing routine is used which erases only those parts of a list that are owned. This system obviously achieves the desired result only so long as no list is erased that owns a part that has been borrowed by some other list not yet erased. In some applications (Gelernter's, in particular) this restriction may be easy to satisfy. In other applications (our own, in particular) avoiding its violation is so difficult that the method is quite useless.

The method that we have devised consists, briefly, in allowing the arbitrary interspersion (in lists) of words containing reference counts. Viewed in terms of the conventional diagrams of lists, such a reference count is a tally of the number of arrows pointing to the box containing the

[1] This will of course depend on the computer, the application, the skill of the programmer, and other factors. Our statement is based on estimates we have made for the IBM 7090, which indicate that total reclamation time may even vary inversely with the number of registers reclaimed.

[2] Our estimates suggest that, even at a level of half-utilization of memory, execution time per word reclaimed would be approximately three times as great for McCarthy's method as compared with our own.

[3] This application is in the development of a program for a modified form of A. Tarski's decision method for the elementary theory of real closed fields. See [1] and [5]. In our adaptation all formulas are normalized and arithmetized. Further, floating-point calculations are used extensively in heuristic testing of the satisfiability of various form formulas.

[4] McCarthy's method can be modified by setting aside a block of memory and establishing a one-to-one correspondence between the bits in this block and the words in the remaining memory. This modification eliminates the awkwardness of arithmetic, but takes a further toll in the speed of reclamation.

reference count and emanating from boxes in the same or other lists.[5] A two-bit type code then appears in each location word, one value of which serves to identify a reference count as such. The obvious convention is adopted that, unless a word contains a reference count, there is exactly one arrow pointing to it. A reference count of one is thereby redundant but otherwise harmless.

Before proceeding to describe our system in detail we wish to establish some definitions and notation pertaining to lists. Besides facilitating the present discussion we believe these definitions will be of interest in themselves. Among other things they may serve to sharpen the useful distinction between lists and various representations, or methods of representation, thereof in computers, and provide some terminology and notation for better communication.

Definitions and Notation

Given an arbitrary set S, we wish to define, as a mathematical entity, independent of computer considerations, the class $\lambda(S)$ of all lists over S. Before proceeding with the definition a few preliminaries are required.

If S is an arbitrary set, we denote by $\sigma(S)$ the class of all finite sequences of elements of S. The standard mathematical notation for such a sequence of length $n \geqq 1$ is $(a_1, a_2, \ldots, a_n)$. Such a sequence can be identified with the n-tuple usually denoted in the same way. This implies, in particular, that one may have $a_i = a_j$ for $i \neq j$. We include in $\sigma(S)$ also the unique finite sequence of length 0, the null sequence. The null sequence can be identified with the null set and thereby denoted by Λ. For the sake of uniformity we shall denote it by ( ).

We can now define $\lambda(S)$, the class of all lists over S, as the smallest class T which contains $\sigma(S)$ and is closed with respect to σ(i.e., $\sigma(T) \subseteq T$). We can alternatively define $\lambda(S)$ by induction. Let $S_0 = S$ and, for $i \geqq 0$, define

$$S_{i+1} = \sigma\left(\bigcup_{j=0}^{i} S_j\right).$$

Then clearly

$$\lambda(S) = \left(\bigcup_{i=1}^{\infty} S_i\right).$$

This second definition makes it possible to define the order of a list. If A is a list over S then we define the order of A (relative to S) as the least $i \geqq 1$ such that $A \in S_i$. It will be convenient to refer to the elements of S as atoms.

To illustrate these definitions, let S be the set of all integers. Then the following expressions denote lists over S of the specified orders.

List	*Order*
( )	1
(2, 1, 1)	1
(( ))	2
((4,1), (3,1,2), (2))	2
(5, (2,5), (3), ( ))	2
((7,9), 4, (4, (3,2), 4))	3

Computer Representation of Lists

We shall describe our method for the representation of lists as applied to the IBM 704, 709 or 7090. It will be easy to adapt the method to many other computers.

Since these computers have word lengths of 36 bits, it is natural to take S to be the set of all "words", that is, all sequences of 36 zeros and ones. The atoms may then themselves be regarded as the representations of various types of objects, e.g., characters, strings of 6 (or fewer) 6-bit characters, signed or unsigned integers, floating-point numbers, etc., depending on application and context.

In addition to providing a solution to the problem of overlapping and erasure, the method of representation which we shall describe has two other merits. First, the list represented is uniquely determined, given the location of the first word of its representation. Second, a storage location may be saved for each atom requiring 15 or fewer bits (i.e., having at least 21 leading zero bits). The first of these is important in that it enables the use of general purpose routines (e.g., for erasure or input-output) which operate on arbitrary lists without supplementary information about their structure.[6]

As in Gelernter's method, each list is represented in a set of location words and data words. Data words contain atoms and consist of a single field of 36 bits. Location words consist of 5 fields:

sign field (one bit)
prefix field (bits 1 and 2)
decrement field (bits 3–17)
tag field (bits 18–20)
address field (bits 21–35)

The sign field and tag field are not used. The prefix field contains the type code. The address field contains the location of the next location word, if there is any, otherwise it contains zero. The decrement field contains any of four things, determined as follows by the type code:

Type Code	*Decrement Field*
0	Location of an atom
1	Atom (last 15 bits, the first 21 being zero)
2	Location of a list
3	Reference count

The set of locations containing a representation of a list contains an unique first location which determines all other locations. We shall therefore speak of "the location of a representation of a list." We define this by induction on the order of the list.

Let A be a list of order k, and let L be a location. If A is the null list (whence $k = 1$), then L is the location of a

[5] There will be no such arrow pointing to the first register of a list which is not a part of any other list. There will, however, be an arrow pointing to it from some program register (base register in McCarthy's terminology) and, strictly speaking, such arrows are also counted.

[6] Gelernter's system shares the second of these features and would have the first feature also except for the absence of a bit to distinguish an address field containing an atom from one pointing to an atom.

representation of A if and only if $L = 0$. Otherwise, let $A = (A_1, \ldots, A_n)$ with $n \geqq 1$. Then each A_i is either an atom or a list of order less than k (if $k = 1$ each A_i is an atom). L is the location of a representation of A in case the following conditions are satisfied (we use as notation $w(L)$ for the word stored at location L, $p(L)$ for the prefix field of $w(L)$, $d(L)$ for the decrement field of $w(L)$, and $a(L)$ for the address field of $w(L)$):

(1) There is a sequence of locations $L_1, L_2, \ldots, L_m$, all different from zero, such that
  (a) $L = L_1$
  (b) $a(L_i) = L_{i+1}$, for $1 \leqq i < m$
  (c) $a(L_m) = 0$

(2) If $L_1', L_2', \ldots, L_r'$ is the subsequence of the L_i for which $p(L_i) \neq 3$, then the following hold:
  (a) $r = n$
  (b) If A_i is an atom, then either
    (i) $p(L_i') = 0$ and $w(d(L_i')) = A_i$, or
    (ii) $p(L_i') = 1$ and $d(L_i')$ consists of the last 15 bits of A_i, the first 21 being zero.
  (c) If A_i is a list, then $p(L_i') = 2$ and $d(L_i')$ is the location of a representation of A_i.

Use of the Method

We have now described completely our method for the representation of lists, but several remarks are in order concerning its use.

When borrowing a list there are two cases to consider according as the location, L, of the representation of the list to be borrowed contains a reference count or not. If it does, it is only necessary to increase this count by 1. If it does not, the necessary procedure is to obtain a new location, L', from the list of available storage, store the contents of L into L', and replace the prefix, decrement and address fields of the word at L by 3, 2, and L', respectively. It is convenient to have a special routine to choose the appropriate procedure and carry it out.

When, during the erasure of a list, a word is encountered containing a reference count, there are again two cases.[7] If the count is greater than 2, it is decreased by 1 and erasure stops.[8]

If the count is 2 the procedure is to store the contents of $L' = a(L)$ into L, where L is the location containing the reference count, return L' to the list of available storage and erasure stops.[8] We employ a simple general-purpose routine for the erasure of an arbitrary list.

In borrowing there are two situations of common occurrence. In one case it is desired to borrow an entire list (e.g. for use as one term of the list being generated). In the other case it is desired to borrow only the tail $(a_i, \ldots, a_n)$ of a list $(a_1, \ldots, a_i, \ldots, a_n)$, treating it (as one may) as if it were an entire list. The tail borrowed may be used, for example, as the tail of the list being generated. This latter case arises, for example, in certain merging operations. It is possible because an arbitrary interspersion of reference counts is allowed. However, for some applications it may be desirable to simplify by adopting the convention that a reference count may occur only in the first location of a list. A tail may not be borrowed then, but instead the individual terms of the tail are borrowed.

A disadvantage of our system is that atoms cannot be borrowed. Often this will not be a significant encumbrance. When it is, it can be partially resolved by uniformly replacing each atom, a, by the list (a) whose only term is a. This is entirely satisfactory if the borrowing frequency for atoms is moderately large.

The method of representation which we have described is predicated on a computer word too small to contain three storage addresses. If the words will hold three address fields (as in the IBM 7030), then one of these fields may be used for a reference count in each location word.

[7] On the basis of an earlier remark, one may assume that a reference count of 1 never occurs. Otherwise a third case arises in which the procedure is to simply erase a location containing a reference count of 1 and proceed from there.

[8] More precisely, erasing is discontinued in the appropriate direction. Other parts of the list may still remain to be erased.

REFERENCES

1. Collins, G. E. Tarski's decision method for elementary algebra. *Proc. of the Summer Institute of Symbolic Logic*, 1957, pp. 64–70.
2. Gelernter, H.; Hansen, J. R.; and Gerberich, C. L. A Fortran-compiled list processing language. *J. Assoc. Comput. Mach.* 7 (1960), 87–101.
3. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3 (1960), 184–195.
4. Newell, A.; Shaw, J. C.; and Simon, H. A. Empirical explorations of the logic theory machine. *Proc. of the 1957 Western Joint Computer Conference*, pp. 218–230.
5. Tarski, A. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.