Palo Alto Research Center

Compact Encodings of List Structure

By Daniel G. Bobrow and
Douglas W. Clark

Compact Encodings of List Structure

by Daniel G. Bobrow and Douglas W. Clark

CSL-79-7 JUNE 1979

Abstract: List structures provide a general mechanism for representing easily changed structured data, but can introduce inefficiencies in the use of space when fields of uniform size are used to contain pointers to data and to link the structure. Empirically determined regularity can be exploited to provide more space efficient encodings without losing the flexibility inherent in list structures. The basic scheme is to provide compact pointer fields big enough to accommodate most values that occur in them, and to provide "escape" mechanisms for exceptional cases. Several examples of encoding designs are presented and evaluated, including two designs currently used in Lisp machines. Alternative escape mechanisms are described, and various questions of cost and implementation are discussed. In order to extrapolate our results to larger systems than those measured, we propose a model for the generation of list pointers, and test the model against data from two programs. We show that according to our model, list structures with compact *cdr* fields will, as address space grows, continue to be compacted well with a fixed width small field. Our conclusion is that with a microcodable processor, about a factor of two gain in space efficiency for list structure can be had for little or no cost in processing time.

CR Categories: 3.69, 4.34, 4.6, 5.6

Key words and phrases: compact encoding, list structure, Lisp, list structure regularity, linearization, Lisp machine

To appear in ACM *Transactions on Programming Languages and Systems*

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California
94304

1. Introduction

Most implementations of list structures use fields of uniform size to point to data and to link the structure. An empirical study of the use of list structure in Lisp [Cla77] indicates that certain values of these fields are used far more often than others. This non-uniformity can be exploited to obtain *compact encodings* of list structure without sacrificing their generality in representing structured data. In these encodings a small field is used to represent the most common values, and an *escape mechanism* provides access to full-size pointers when they are needed. In this paper we explore this idea in detail by constructing and evaluating several encoding schemes and escape mechanisms. Although we discuss the execution time costs of these designs, we concentrate in this paper on careful evaluations of their savings in space.

The empirical data upon which these encoding designs and evaluations are based are reported in full in [Cla77, Cla79]. We summarize some of those results here. The list structures in existence at the end of a typical run of five large Lisp programs (about 50,000 cells each) were measured to determine the frequency of occurrence of pointer values in *car* (usually the data field) and *cdr* (usually the link field) of each list cell. Some related dynamic measurements were also made, both to determine the relative frequency and arguments of basic operations, and to verify that list structures measured at the end of the run were typical of those existing at various points during the run. The structures examined were just those used as data for the programs, and not those used to represent the programs themselves. Deutsch has shown that great coding efficiency can be obtained for programs by compiling them into a carefully designed instruction set [Deu73b]; therefore, we will not address the problem of compacting program list structure in this paper. Our data came from the following five programs: CONGEN (called "STRGEN" in [Cla77]), a chemical structure generator [Smi74]; NOAH, a planning program [Sac77]; PIVOT, a program verifier [Deu73a]; SPARSER, the parser in a speech understanding system [Bat75]; and WIRE, a wire-listing program used at Xerox PARC. All five are written in Interlisp [Tei74], a sophisticated Lisp system running on the PDP-10 computer under the Tenex operating system [Bob72].

Table I shows how pointers in *car* and *cdr* were distributed among the data types of Interlisp. *Atoms* are Lisp's symbols or identifiers, and *small integers* are those in the range ±1536. NIL is a special atom normally used in *cdr* to indicate the end of a list. There is considerable agreement among the programs, especially in *cdr*.

TABLE I
Distribution of data types in list cells (numbers are percentages)

	CAR		CDR	
	range	mean	range	mean
NIL	1.0-5.6	3.1	24.2-26.6	25.1
lists	23.3-31.5	28.6	66.7-74.8	72.5
non-NIL atoms	34.2-58.4	45.9	0.5-4.5	1.5
small integers	4.1-34.7	19.5	0.1-1.0	0.5
others	1.6-7.1	2.9	0.0-1.1	0.4

List pointers most often pointed only a short distance away in the address space of list cells. This is shown in Table II. In both *car* and *cdr*, an offset of one was most common, and there was a strong decline as distance increased. This is a reflection of the fact that free lists tend to start out ordered, and cells that point to each other are most often created close in time [Cla77]. This very strong regularity is the reason that we will most often represent list pointers as cell-relative offsets in the encodings of this paper. Some encodings will use *page*-relative offsets, which also benefit from the regularity shown in Table II.

Table II
Cumulative distribution of relative offsets of list pointers

	CAR		CDR	
offset	range	mean	range	mean
±1	19.4-36.3	29.8	53.7-75.8	61.3
±2	26.2-46.1	38.7	58.1-76.6	65.7
±4	35.4-53.6	46.9	66.8-79.7	72.8
±8	44.5-66.8	58.4	72.5-86.6	79.1
±16	47.7-72.8	64.0	77.5-92.6	84.5
±32	49.8-76.2	67.7	81.4-93.8	87.8
±64	51.6-80.3	71.2	84.7-94.9	90.0
±128	53.5-84.0	73.8	87.1-96.3	91.6
±256	55.2-87.0	76.2	88.9-97.0	92.8
others	13.1-44.8	23.8	3.0-11.0	7.2

Pointer distances become dramatically smaller when list structures are *linearized* [Fen69, Min63]. Linearization is the rearrangement of a list in memory so that *cdr* points to the next sequential location whenever possible, i.e., when not prevented by the sharing of list cells among several pointers. Algorithms for doing this differ chiefly in how they handle the *car* sub-structures encountered during the *cdr*-first traversal: some linearize these *car* lists in FIFO order (e.g., [Che70, Bak78]; others use LIFO order (e.g., [Min63, Fen69, Cla76a]). In this paper we use exclusively the latter approach. Linearization is normally thought of as a concomitant to garbage collection, but linearized lists can also be created during ordinary computation by appropriately designed Lisp primitives (e.g., *read*, *list*, and *append*) if the free-list is ordered. Linearizing the list structures of the five programs made 99 percent of list-pointer *cdrs*, on average, point to the next cell, while improving the *car* pointer distributions slightly [Cla77].

Pointers to non-NIL atoms were found to follow, roughly, Zipf's law, which models the occurrence of words in natural language text as well as many other phenomena [Knu73, p. 397; Zip49]. According to Zipf's law, the i^{th} most common item in a collection occurs with frequency proportional to $1/i$. Table III shows the frequency distributions of atom pointers, grouped in binary decades of frequency rank. (CONGEN makes unusually heavy use of a small number of atoms and is excluded from Table III; atom encodings in this paper will always do better with CONGEN than with the other four programs.) The number of atoms in each program was between 2477 and 4711; this variation will, of course, distort inter-program comparisons based on frequency rank. The column labeled "scaled mean" contains the mean frequency in each binary decade after scaling to make each program's 2047th atom be its last. Zipf's law predicts that the numbers in this column should equal the last column of Table III, approaching a constant from above as rank increases. Instead the "scaled mean" column approaches a slightly larger constant from below; thus Zipf's law overestimates the frequency of the programs' most common 20-30 atoms, but does reasonably well for the rest.

Table III
Frequency distribution of pointers to non-NIL atoms

frequency rank	range	mean	cumulative mean	scaled mean	Zipf prediction for scaled mean
1	2.1-8.1	4.4	4.4	4.7	12.2
2-3	3.8-7.6	5.6	10.0	5.9	10.2
4-7	5.3-9.0	6.6	16.6	7.0	9.3
8-15	5.4-8.1	7.1	23.7	7.5	8.8
16-31	6.7-10.8	8.6	32.2	9.0	8.6
32-63	6.8-14.3	10.2	42.4	10.8	8.5
64-127	7.9-11.9	10.4	52.8	11.1	8.5
128-255	9.7-11.7	10.7	63.6	11.4	8.5
256-511	8.7-14.4	11.1	74.6	11.8	8.5
512-1023	7.5-11.3	10.3	84.9	11.0	8.5
1024-2047	6.4-12.6	9.3	94.1	9.9	8.5
2048-4095	1.5-10.7	5.3	99.4	---	---

The rest of this paper is organized as follows. The basic ideas of list structure encoding are illustrated by several simple examples in Section 2, which includes a discussion of some general principles illustrated by the examples. Escape mechanisms are very important to the success of compact encodings, and are discussed in detail in Section 3. Some of these mechanisms are used by the encoding designs in two Lisp machines [Deu73b, Deu78, Gre74, Baw77], which are presented and evaluated in Section 4. Section 5 explores some of the costs of using compact encodings in list processing systems. Finally we discuss in Section 6 how well these ideas might stand up as address spaces get dramatically larger.

2. Simple examples of space-conserving encodings

Suppose that we want to represent list structures using 18-bit cells instead of the 36-bit cells used in PDP-10 Interlisp. In this section we will informally design a set of simple encodings illustrative of the principle of matching an encoding design to the empirical data. Three encodings will be specified, corresponding to three divisions of the available 18 bits between *car* and *cdr*: 16-2, 12-6, and 9-9. Our bias towards giving more bits to *car* is due to the large number of atom pointers found in *car*.

We will assume the use of one of the simplest uniform escape mechanisms to allow remote storage of pointers that do not fit into the fields. The mechanism we have chosen for these examples is to reserve a single bit-pattern for the field (say all 1's) to indicate that the desired value can be found in a global hash table. The key in the hash table is the address of the cell that should have contained the pointer. This idea is called "hash linking" by Bobrow [Bob75]. (Since *car* and *cdr* could both have the escape value, one more bit needs to be hashed along with the address of the cell.)

2.1 Specification

If a compact field is b bits wide, there are 2^b *codewords* or *codes* available, one of which, in these examples, is reserved for the hash-link escape. The remaining 2^b-1 must be divided among the most common values found in the field. For the encoding of the *cdr* field, we note that one codeword should clearly be used for NIL, the atom that accounts for about one-fourth of all *cdrs* (Table I). We can divide the remaining patterns between relative offsets of list pointers, and pointers to the most common nonlist objects. But the latter are rare in *cdr*, so the allocation of some codes for the most common atoms or arrays or whatever would probably not be worthwhile. In the 2-bit *cdr* case, the best we can do is represent list pointer offsets of +1 and -1, the pointer NIL, and the escape. In the 6-bit case, we can use the 62 codewords that remain after representing NIL and the escape to represent offsets of -31 to +31. Similarly, for the 9-bit *cdr*, we will represent list pointer offsets -255 to +255. (Our choice of a symmetric window for these relative offsets is arbitrary but simple, and matches our data fairly well; more highly-tuned encodings might position the window asymmetrically. In a linearized system, of course, most offsets would go in one direction.)

The allocation of *car* codes is more difficult, since different data types abound in *car*. All three encodings need one code for the escape and also one for NIL (3.1 percent of *cars* on the average). The popular data types in *car* are lists, atoms, and small integers. For the 9-9 encoding, the remaining 510 codewords will be divided as follows: 348 for the most common atoms, 100 for the most common small integers, and 62 for list pointer offsets -31 to +31. For the 12-6 encoding we will keep the 62 codes for lists, and increase those for atoms and small integers to 3007 and 1025, respectively. If the *car* field is 16 bits wide, the escape will never be needed: four of the programs have fewer than 2^{16} total cells, and there is sufficient regularity in *car* list pointers that the offset representation will easily bring down the total number of different *cars* in the fifth program to less than 2^{16}. Almost any sensible choice--for example, about 5000 codes for atoms, 3000 for small integers, 52000 for lists, and 500 for other data types--will accomodate all existing values in all five programs.

2.2 Evaluation

We have now specified several simple encoding schemes for *car* and *cdr*. What savings in space, if any, will these designs have over the Interlisp implementation of list cells? Let p and p' be the fractions of *cars* and *cdrs*, respectively, that fit into the small fields of size b and b', and let c and c' be the numbers of bits used by escape values of *car* and *cdr* over and above the bits in the escape codeword itself. Then the average number of bits required is $b+(1-p)c$ for *car* and $b'+(1-p')c'$ for *cdr*.

Evaluation of the 12-6 encoding. Values of the average p and p' can be found using Tables I-III and more detailed data from [Cla76b, Cla77]. Consider first p'. All NILs will fit into the 12-6 encoding, accounting on the average for .251 of all *cdrs*. List pointers with distances between -31 and +31 account for .87 of list *cdrs*; lists are .725 of *cdrs*; therefore .87*.725 = .631 is the fraction of all *cdrs* that are lists representable by a short pointer. The value of p' is thus .251+.631 = .882. Thus .118 of all *cdrs*, those which are neither nearby lists nor NIL, need the escape value and another c' bits each. The average size of a *cdr* is $6+.118*c'$ bits.

Notice that c' (and also c) can be large: if a hash table is used, the simplest possible scheme requires an 18-bit key (to resolve collisions) and the 18-bit true value of *cdr*. With $c'=36$, the average size of a *cdr* in the 12-6 encoding is 6+.118*36 = 10.2 bits. (In fact, c' should be larger still, to account for the hash table's not being completely full; on the other hand we will show in Section 3 that it is easy to do better than $c'=36$.)

We now make a similar calculation for average *cars*, again using Tables I-III. The atom NIL accounts for .031 of them; list pointers with distances in the range -31 to +31 account for .677 of list *cars*, and .194 of all *cars*. If we assume that all small integers used are actually in the range -512 to +512--a reasonable and supportable approximation--then .195 of the compact *cars* will specify small integers. The programs average 2996 atoms each, so the 3007 available codes will handle all of the atoms, for another .459 of the *cars*, on average. This gives p = .031+.194+.195+.459 = .879, leaving a remainder of .121 that will use the escape mechanism. The average size of a *car* in this encoding is then $12+.121*c$ bits. If c is 36, the average bit commitment comes to 16.4. Thus the average size of a *cell* in the 12-6 encoding is 10.2+16.4 = 26.6 bits, saving more than 25 percent in space over the standard 36-bit encoding.

Evaluation of the 9-9 encoding. Evaluation of the 9-bit *car* is slightly more difficult, since we can no longer accomodate all of the atoms and small integers. We must choose the most common 348 atoms and 100 small integers. On average, the top 348 atoms account for .335 of all *cars*, and the top 100 small integers for .123 of them. The 62 list codes get us, as before, .194 of the *cars*. The value of p is .031+.194+.123+.335 = .683. With c = 36, the average size of *car* for the 9-9 encoding is 20.4 bits.

The 9-bit *cdr* will get all NILs and all list offsets in the range -255 to +255, giving, for the average statistics, p' = .251+.673 = .924. With c' = 36, the average *cdr* is 11.7 bits wide. The average cell size is 20.4+11.7 = 32.1 bits.

Evaluation of the 16-2 encoding. As described above, the 16-bit encoding of *car* will easily handle all *cars* in each of the programs, giving p = 1 and an average size of 16 bits.

If the *cdr* field is only 2 bits wide, we can fit into the encoding just the two list offsets +1 and -1, the atom NIL, and the escape. This gives $p' = .251+.444 = .695$ and an average *cdr* size of 13.0 bits. The average cell size for this encoding is $16+13 = 29$ bits.

These encodings were designed with the average statistics of Tables I-III in mind. It seems reasonable to repeat the average bit calculations for the five programs' individual statistics, particularly in view of the similarity among them. These results are shown in Tables IV, V, and VI. The statistics in the row labelled "Average" represent the encoding effectiveness of a program with average statistics; this is *not* the same as the average of the other five numbers in each column.

If linearization in the *cdr* direction is done on the list structure, then about 97% of all *cdrs* can be encoded in two bits [Cla77], with the escape adding less than .9 bits to the total cell size. The last column of Tables IV, V, and VI show approximately what the cell size would be if linearization were done.

TABLE IV
Evaluation of the 16-2 encoding

	CAR p	size	CDR p'	size	CELL size	linear CELL size
CONGEN	1.00	16.0	.736	11.5	27.5	18.9
NOAH	1.00	16.0	.666	14.0	30.0	18.9
PIVOT	1.00	16.0	.624	15.5	31.5	18.9
SPARSER	1.00	16.0	.645	14.8	30.8	18.9
WIRE	1.00	16.0	.813	8.7	24.7	18.9
Average	1.00	16.0	.695	13.0	29.0	18.9

TABLE V
Evaluation of the 12-6 encoding

	CAR p	size	CDR p'	size	CELL size	linear CELL size
CONGEN	.914	15.1	.941	8.1	23.2	22.0
NOAH	.894	15.8	.862	11.0	26.8	22.7
PIVOT	.820	18.5	.845	11.6	30.1	25.4
SPARSER	.871	16.6	.840	11.8	28.4	23.5
WIRE	.859	17.1	.944	8.0	25.1	24.0
Average	.879	16.4	.882	10.2	26.6	23.3

TABLE VI
Evaluation of the 9-9 encoding

	CAR p	size	CDR p'	size	CELL size	linear CELL size
CONGEN	.869	13.7	.957	10.6	24.3	23.6
NOAH	.750	18.0	.911	12.2	30.2	27.9
PIVOT	.581	24.1	.882	13.2	37.3	34.0
SPARSER	.666	21.0	.897	12.7	33.7	30.9
WIRE	.682	20.4	.972	10.0	30.4	30.3
Average	.683	20.4	.924	11.7	32.1	30.3

2.3 Discussion

There are several interesting observations to be made about Tables IV-VI. *Cdrs* are more successfully encoded than *cars*, for all three encodings. This is, of course, due to the much greater regularity found in *cdr*; what especially hurts the *car* figures is the large number of pointers to a large number of atoms. Some programs even have a bigger "encoded" *car* than the 18 bits they started out with. The innate compressibility or encodability of *car* and *cdr* can be measured precisely by calculating *pointer entropy* [Cla77]. Entropy serves as a loose lower bound on the efficiency of all single-symbol encodings. The entropy of *car* pointers for the five programs was between 6.4 and 10.3 bits; CONGEN had the lowest *car* entropy, and PIVOT the highest, as is suggested by Tables V-VI. *Cdr* entropy ranged from 3.3 bits for WIRE to 5.2 bits for PIVOT, again agreeing with the results of Tables IV-VI.

Because pointer decoding depends on the type of the object pointed to (except for escapes), that type must be a function of the codeword alone. (This is a useful property in Lisp systems generally [Tei74] since it allows the common type checks such as *atom*(x) and *listp*(x) to be performed without actually looking at the object referenced.) Therefore in deriving an encoding we must allocate a fixed number of codewords (possibly 0) to each data type. The optimal strategy in allocating these codewords is to "balance" the encoding design so that the marginal increase in pointers accounted for is equal across all data types. That is, the most infrequent codeword for each type should be used about equally often. Because the frequency distributions are discrete, not continuous, exact equality may not be achievable, but it should be the goal of a design. An example of an encoding designed like this may be found in [Cla76b]. Allocating codewords this way is equivalent to simply sorting the things that occur in (say) *car* and picking the most common, regardless of data type. Grouping these in the codeword space should be done so as to maximize implementation efficiency. If the sorting order changes much over time, then one must also include the cost of maintaining the information about the pointer interpretation; for example, if the set of most commonly referenced atoms changes composition radically during the course of a computation, then any assignment of interpretation made at one time will lead to less than optimal encoding at another time.

Is there an "optimal" compact field size for a fixed set of list structure statistics? Tables IV-VI show that all five programs do better with a 6-bit *cdr* than with either a 2-bit or a 9-bit one; this suggests that the optimal size is between 2 and 9. There is a trade-off between field size and escape cost: the optimal field size should be chosen so that for field size b, escape probability $(1\text{-}p)$, and constant escape cost c, the expression $b+(1\text{-}p)c$ is minimized. Since $(1\text{-}p)$ is a decreasing function of b (bigger field, fewer escapes), there is, for fixed c, a unique minimum. The best field size can be found by increasing b a bit at a time, and stopping when $b+(1\text{-}p)c$ increases rather than decreases.

Tables IV-VI suggest that for all five programs, an escape cost of 36 bits dictates a compact *cdr* size of more than 2 and less than 9 bits. The tables do not *prove* this because the encoding designs did not optimally assign codewords to data types, as discussed above; the bad performance of the 9-bit *cdr* might be due to a poor codeword assignment.

The performance on linearized list structure indicates that the minimal 2-bit *cdr* is quite effective, and is, in fact, "optimal" in the above sense. The average escape cost $(1-p')c'$ is less than one bit, so increasing the compact *cdr* to 3 bits would make things worse. If a large proportion of the data were static and could be linearized, then it is clear that using a 2-bit *cdr* field is very efficient. Data from [Cla79] indicate that once list structure is linearized it tends to stay that way (that is, programs do not disturb data structures much after creating them). This is a basic assumption underlying the encoding used for the MIT Lisp machine, whose encoding we describe in Section 4.

3. Escape mechanisms

There are a variety of mechanisms for handling the small fraction of pointers that will not fit into a compact encoding. The possibility of using a single global hash table was briefly considered in Section 2; here we will discuss that method, and others, in greater detail. There will, in general, be two important criteria by which these schemes must be judged: first, the cost in extra bits per escape pointer, and second, the cost in execution speed of using a particular method. In the description below, a *long pointer* is one that can address any object.

3.1 Indirect tables

Suppose that on each page (or other convenient block of contiguous list storage), all the long pointers are grouped together into a table. Some number of codes in the compact field could be used as indices into this table, indicating that the pointer encoded in that field is to be found indirectly, that is, in the table. This scheme costs a proportion of the codeword space sufficient to address the table, and a fixed commitment of space for the table itself.

The size of the table is an issue. Ideally it would be just big enough to accomodate most of the long pointers occurring on the page. If the table is too small, it will fill up before the page itself, creating a problem with several possible solutions: the page might simply be declared "full", thus wasting the unoccupied cells; or an alternative (and more costly) escape mechanism, such as a hash link, could be used when new cells were put on a page whose table was full. The latter method would be absolutely essential if a short pointer on a "full" page were to be changed with *rplacd* into a long pointer; this drawback forbids the use of indirect tables alone.

Because the allocation of tables and the means of addressing them are fixed, it becomes possible to make the individual entries have different sizes. For example, in an 18-bit address space, a full 18-bit pointer would probably be needed only rarely. Perhaps a 9-bit or 12-bit pointer would do much of the time. The indirect table could then contain various sizes of entries, arranged so that the size of the entry is a function only of the index.

3.2 Indirect relative pointers

An immediate generalization of the indirect table idea is to put the long pointer in *some* nearby word on the page and point to it with a short pointer. Thus the same words on a page can be used either for list cells or for indirect pointers, eliminating the problem of having to fix the size of the indirect table. However, this technique has a problem in the (presumably rare) case that after allocation of a new cell needing an escape, the next word on the free-list is too far away to address with a short codeword. The *rplacd* problem becomes less severe, but can still cause trouble, and there remains the potential problem that a compact cell may not be big enough to store a long pointer.

A fraction of the available codewords must be assigned to indirect pointers. The most obvious way to assign these codes is to have an indirect bit in the short field; but this consumes half of the available codes--almost certainly not the best way to use them. It might be better to allow only a small number of short indirect pointers, and to rely on a different escape mechanism (such as a hash link) if an indirect pointer cannot be used.

Note that if a compacting garbage collector is used and the free-list is kept ordered, these indirect pointers need only be able to point in the direction of the free-list links. Every creation of one of these short indirect pointers will involve two consecutive cell-allocations, and the second of any two consecutive allocations will always have a larger address than the first.

A useful specialization of the indirect relative pointer is to require that the long pointer be stored in the *next* cell. This idea is used in the MIT Lisp machine. If a compacting garbage collector is used then when *cons* is done, the next cell on the free-list will be adjacent to the cell just allocated, and available for the escape. However, if *rplacd* is done on the first cell after one or more additional *cons*es, another escape mechanism will be needed.

3.3 Hash links

In Section 2 we described a use of Bobrow's [Bob75] *hash links* as an escape mechanism. In this simple form, if the compact *cdr* (or *car*) field of cell x contains a particular escape codeword, the true value of $cdr(x)$ (or $car(x)$) is found in a hash table, where x is the key. Bobrow points out that one might have a separate hash table on each page; in that case the stored key would only need to be an index onto that page. He also proposes a single global table for all hash links; there the key must be a full-size pointer.

If several escape codewords can be used, the value stored in the hash table can be extended with additional bits, as Bobrow points out. For example, in a short field of b bits, imagine that if the high order r bits all equal 1, the remaining b-r bits are to be used as an extension of the value retrieved from the hash table. One might also consider using different escape values for different tables. A table for each data type would seem to be a good idea if the long pointers associated with different types have different lengths.

Another space-saving possibility is to have one hash table in which no keys are stored, only a bit to indicate whether this entry was made without a collision. Should a collision occur when a new entry was being added, the new entry would be put in a second table in which collisions were allowed and keys stored, and the "unique" bit in the first table would be cleared. On lookup, for entries marked as unique (no collision of hash on entry), the lookup procedure can assume the match of the key, and use the entry directly. For entries marked as non-unique, the collision table must be checked first. If a matching key is found in an entry there, then that is the correct one. Otherwise the non-collision table entry is correct. If the table sizes are chosen properly, most entries will be found in the non-collision table.

Various familiar hash table costs must be faced. There is first the space overhead of empty cells in the table, possibly maintained for lookup efficiency [Knu73]. Second, there is the problem of rehashing entries into a new larger table when the growth of list structure requires it. This could be done at garbage collection time. And third, there is the computation of the hash function itself. This could be made quite fast with hardware or microcode assistance, and need not involve additional references to main storage. Numerous other aspects of hash table implementation are discussed by Knuth [Knu73].

Two other potential problems must be considered. One involves local tables, the other, global ones. In the case of a local table of fixed size, a page of list cells that has an unusually high number of escapes can simply not be stored. If the table is full (and if a *global* table is not available), no more cells can be put on the page. This potential waste of non-table bits on the page must be counted in the average cost of an escape.

The problem with global tables is that they must be *large*, and therefore may not be in core when they are needed. This is another time penalty associated with the dynamic use of a hash link: there is some nonzero chance that retrieving the desired value will cause a page fault. Schemes in which the hash table is somehow locked into main memory bear a cost in dynamically available memory space.

3.4 "Invisible" cells

All of the escape schemes discussed so far have the property that *car* and *cdr* are independently encoded. Relaxing this property yields an interesting and useful escape mechanism that we call *invisible cells* after Greenblatt [Gre74]. In an invisible cell a special *cdr* codeword indicates that this is not a "real" cell, and that the real cell can be found by following *car*, which in this case will be called an *invisible pointer*. (*Car* and *cdr* could be reversed, but because *cdr* is usually smaller, this way is better.) The invisible pointer will most commonly point to a cell with unencoded *car* and *cdr*.

SWYM [Han69] has the earliest use we know of invisible pointers in the compact encoding of list structures. SWYM provided a full size field for the data item, but always assumed that a list continued in the next contiguous cell in memory. (This assumption required SWYM to prohibit the *rplacd* operation.) When this was not the case, a special bit was set indicating that the data portion of the cell contained the location of the list continuation.

Using an invisible cell is a way to deal with the *rplac* problem. When a compact cell is *rplac*'d in a way with which the other available escapes cannot cope, the entire cell can be made "invisible" and relocated to a more convenient spot. Pointers that go through an invisible cell can be updated to their new value either when they are traversed or during garbage collection. This is called "snapping" the pointer by Greenblatt [Gre74]. Snapping the link on invisible cells in a system forces the test for pointer equality (EQ in Lisp) to check the contents of the cell since two pointers should be considered EQ if one is a pointer to an invisible cell, and the other is the contents of that cell. For just this reason the MIT system calls for snapping links only at garbage-collection time [Baw77].

4. Lisp machine encodings

Two existing Lisp machines use compact encodings for *cdr*, one at Xerox PARC [Deu73b, Deu78], and another at MIT [Gre74, Baw77]. Both have a 24 bit virtual address space, and are microprogrammable to provide efficient support for the decoding of compact pointers. Their encoding schemes will be described and contrasted in terms of their efficiency under two sets of assumptions about the linearity of lists, using statistics from Tables I-II and from [Cla77, Cla79].

4.1 Xerox Lisp Machine

The Xerox encoding uses pages containing 128 32-bit list cells. The first cell of each page is used as the head of the free list for that page. The other 127 are available list cells. Each cell is divided into a 24-bit *car* field, and an 8-bit *cdr* field. *Car* is large enough to point to anything in the address space. When $cdr=0$ it is interpreted as a pointer to the atom NIL; if *cdr* is in the range 1-127, then these bits are interpreted as the low order bits of the address of *cdr* relative to the page containing this cell. In the range 129-255, *cdr* is interpeted as a page-relative address of a cell whose *car* field contains a pointer to the actual *cdr* value (an indirect page-relative address). If $cdr=128$ then the cell is "invisible" and *car* points to another cell in which the actual values of *car* and *cdr* can be found. The four possible cases for interpretation of a cell are summarized in Table VII below.

TABLE VII
Xerox Lisp machine encoding

cdr contents	*cdr* interpretation	*car* interpretation	size of cell
0	NIL	Standard pointer	32 bits
1-127	Direct page relative	Standard pointer	32 bits
128	Indicates invisible cell	Invisible pointer	96 bits
129-255	Indirect page relative	Standard pointer	64 bits

To analyze the effectiveness of this encoding we need to consider when a list cell can be 32 bits, 64 bits, and 96 bits. It will be greater than 32 bits in just those cases in which *cdr* points to a list that is not on the page--about 7.3 percent of *cdrs*, on average--or when it is neither a list nor NIL--2.4 percent of *cdrs*. Therefore, about 90.3 percent will take only 32 bits.

A 96-bit cell can only be created when a 32-bit cell *on a full page* has its *cdr* replaced by an off-page list or a nonlist other than NIL. Unfortunately, our measurements do not permit us to calculate precisely how often this would occur. We can, however, make the following approximation. Dynamic measurements [Cla79] show that the number of *rplacds* executed during short runs of three programs is about half the number of cells referenced. (By *rplacd* we mean any write of a cdr field, not just those explicitly specified by the programmer. Many such writes are imbedded in Interlisp list-manipulation functions. It should also be recalled that list-structure representations of programs, which are changed very rarely, are excluded from these measurements.) If we assume that no cell gets *rplacd*'d more than once, this implies that about half of all cells undergo a *rplacd*.

About 5 percent, on average, of all *rplacds* change a NIL or an on-page *cdr* into an off-page list or a nonlist other than NIL [Cla79]. In the worst case *all* such *rplacds* will operate on 32-bit cells on full pages, and therefore about 2.5 percent of all cells would be 96 bits wide, leaving 7.2 percent at 64 bits, and 90.3 percent at 32 bits. In this pessimistic case the average cell size would be

$$(.025 * 96) + (.072 * 64) + (.903 * 32) = 35.9 \text{ bits.}$$

In the best case there will be no 96-bit cells, 9.7 percent 64-bit cells, and the same 90.3 percent 32-bit cells; in this case the average cell size would be 35.1 bits. Our uncertainty about the 96-bit case, therefore, changes the average cell size by less than one bit.

If all lists are linearized then the percentage of *cdrs* that are off-page lists or nonlists other than NIL decreases from 9.7 to about 3.1. After a linearization is done, there will be no 96-bit cells, since these can only be created by subsequent executions of *rplacd*. Thus the average size of a cell after linearization is 33 bits.

4.2 MIT Lisp machine

The MIT machine also uses a 24 bit virtual address space. Each 32-bit word contains a 24-bit *car* field, a 2 bit *cdr* field, and 5 bits of data type information. We include the data type bits in our calculation of encoding efficiency because in the Xerox encoding the data type of a pointer is found using the page number of the pointer and a map of how data types are laid out in the address space. In the MIT encoding, the four possible *cdrs* are: 1) *cdr*=NIL; 2) *cdr* is the next cell; 3) *cdr* is pointed to by the next cell--an indirect cell-relative escape, discussed in Section 3.2; and 4) this cell is "invisible"--*car* points to the "real" cell. The first two cases make a one-word cell, the next makes a two-word cell, and the last makes (most often) a three-word cell.

About 46 percent of cells will have a *cdr* accomodated by the first two cases above (Table I and [Cla77]); this leaves 54 percent that take either two or three words to represent. Again, our data fail us when we try to measure precisely how many cells would take three words. This will happen only when a one-word cell undergoes a *rplacd* and the new *cdr* is neither NIL nor a list in the next cell and the next sequential cell is not available. Assume that the next cell is never available. Then *rplacd* will change a one-word cell into a three-word cell an average of 16 percent of the time; given that about half of all cells have *rplacd* done to them, this means that 8 percent of cells will take three words to represent. Therefore 54-8 = 46 percent will take two words, and the average cell size is

$$(.08 * 96) + (.46 * 64) + (.46 * 32) = 51.8 \text{ bits.}$$

If all lists are linearized, the MIT encoding does much better. There are no three-word cells after a linearization, so the average cell size will be the same as that for the Xerox encoding, 33 bits. If list structure is kept mostly linearized with a copying garbage collector, as the MIT design calls for [Bak78], then space is used very efficiently. But cells created *after* a linearization could be more efficiently encoded if the two relative offsets -1 and +1 were *both* available. Our data indicate that both are frequently used in *cdr.*

On the other hand, these data come only from Interlisp programs and are functions in part of the details of that system's coding of various Lisp functions. The MIT system is different from Interlisp in its underlying implementation; such things as the ratio of *rplacd* to *cons* might therefore be different for it. Strictly speaking, our evaluation of the MIT encoding is valid only to the extent that statistics for the MIT Lisp machine are similar to those we have gathered for Interlisp. In the MIT Lisp machine, many of the standard functions (e.g., *list, append*) are designed to build structures with good *cdr* locality consistent with fast execution and good paging behavior. Thus, our computation of the average size of an MIT Lisp machine list cell may be too high.

5. Costs and implementation questions

Any scheme for encoding and decoding information carries some time penalty. We have thus far looked chiefly at the space side of the time-space trade-off; here we consider the time costs of compact encodings, as well as some other implementation issues.

5.1 Frequency of escape

Perhaps the most important question to ask is how often an escape mechanism is used *dynamically.* We know from the data of Section 1 and [Cla77] how many *static* instances of escape codes there will be for each of our programs. Corresponding *dynamic* data reported in [Cla79] can show how often the time penalty of the escape mechanism will be incurred. In [Cla79] it is shown that static and dynamic data roughly agree: data-type distributions were similar in the two settings, as were the distributions of pointer distances; ranking atoms by static frequency did not do violence to the cumulative distribution of dynamic atom references. This means that in general, the *time* penalty of a proposed escape method will be borne approximately as often as the *space* penalty occurs statically.

5.2 Impact on Lisp primitives

The standard approach to pointers in Lisp--using full machine addresses all the time--is, while wasteful of space, very fast: *car* and *cdr* are simple memory read operations; *cons* is a space allocation followed by a simple write. Under a compact encoding scheme, the primitive pointer-manipulating operations of Lisp become, most of the time, only slightly more complicated. In the rare case that an escape is used, the possibly high costs briefly mentioned in Section 3 must be paid.

Consider *car* and *cdr*. These operations must fetch the appropriate small field and check its type: list, atom, NIL, escape, etc. This can be done quickly in hardware or microcode. Most of the time an escape value will not be found; if one is, more complicated decoding must be done. In the usual case, the true value of *car* or *cdr* can be obtained either by a single addition (for lists or small integers) or a table look-up (for atoms).

The Lisp operations *rplaca* and *rplacd* replace an existing *car* or *cdr*, respectively, with a new value. If compact encodings are used, these operations must first check to see whether the new value can fit into the small field. In the case of lists, this will cost a subtraction and a comparison; in the case of atoms, a table look-up to see if it is one of the common few hundred or so. The patterns of data type replacement by *rplac* operations are discussed in [Cla79]; here we are interested chiefly in how often "small" pointers (compact ones) are replaced by "large" pointers (escapes). That, after all, is the most difficult kind of *rplac* under a compact encoding. The data reported in [Cla79] do not include enough information to calculate how often this happens with *rplaca* (atom and number distributions were not calculated), but there is enough to evaluate the more common *rplacd* operation. If we approximate "small" *cdrs* by NILs plus all list pointers with distance under 32, then in the dynamic runs of [Cla79], the replacement of a small pointer by a large pointer accounted for just 2.8 percent of *rplacds* in CONGEN, 5.5 percent in SPARSER, and 6.6 percent in NOAH. Most *rplacds* (63 to 90 percent) replaced one small pointer by another: an easy operation under a compact encoding.

The final list-manipulating primitive, *cons*, creates a new cell containing two given pointers. Once the cell is allocated, a *cons* looks, for the purposes of this analysis, very much like a *rplaca* together with a *rplacd*, and the same computational work must be done. Notice, however, that the construction of an escape is made easier by the likelihood that nearby cells are unoccupied at the time of the *cons*, and thus are available for indirect escapes. In fact, if a linearizing garbage collector is used, we are guaranteed that at the time a *cons* is done the next cell is unoccupied.

5.3 Atom frequencies

If a compact encoding scheme does not allow all atoms to be referenced directly, some way must be found to discover which are the most common ones, in order to realize the savings afforded by the encoding. The problem here is quite different from the case of lists: whereas we can be reasonably sure beforehand which relative list pointers will be common (namely, the small ones), we have no way of knowing before a program runs exactly which atoms will be frequently pointed to in list structure. The obvious way to find out is to count the references to each atom during garbage collection. The cost of doing this during garbage collection is a small amount of time and enough space enough to store the counts, plus a sort of the counts when the dust settles. After the common atoms have been tabulated, the space can be returned until the next garbage collection.

Notice that if the static frequency distribution of atoms is not stable over time, much effort can be spent in encoding and re-encoding atom pointers to maintain coding efficiency. But these distributions almost certainly change slowly, so recalculating them at garbage collection time, or perhaps more rarely, seems sufficient.

5.4 Stability of linearization

Linearizing garbage collection can greatly benefit a compact encoding, as we have seen. The MIT and Xerox Lisp machine designs both call for linearization to maintain their *cdr* encodings [Bak78, Deu76]. An important question is what happens to linearized list structure between linearizations. An experiment with CONGEN and SPARSER showed that linearized structure is very stable: after a linearization of all lists, each program perturbed its list structure only slightly [Cla79]. It seems likely that *rplacd* (and also *rplaca*) are most often performed shortly after the *cons* that created the cell they change.

6. Extension to larger address spaces

The empirical data used in this paper comes from programs with an effective address space for list structure of about 2^{16} cells. We have suggested on the basis of these data that one might use compact representations of most pointers. A serious issue for the application of these ideas in future systems is how well encoding efficiency will hold up as address space grows. We will be concerned with the encodability of pointers to atoms and pointers to lists. In this section we will argue that a fixed-width compact encoding of *cdr* will, as address space grows, accomodate a fraction of *cdr* pointers that is bounded from below by a nonzero constant. We will also show that a corresponding argument is difficult to make for *car* pointers, both because *car* points most often to an atom, and because list-pointer clustering in *car* is less marked than in *cdr*.

Suppose that we choose to encode the most common k items out of a population of N. ("Items" will be either atoms or list-pointer distances.) Each item i occurs with probability $p(i)$, and

(6-1) $$\sum_{i=1}^{N} p(i) = 1$$

A fixed-width encoding will succeed if the most common k items account for a fraction of all pointers that does not vanish as N increases:

(6-2) $$\lim_{N\to\infty} \sum_{i=1}^{k} p(i) \neq 0$$

If condition (6-2) holds then the cost of an increasing address space will eventually be borne in the size of the pointer delivered by the escape mechanism, rather than in the size of the encoded field itself. For an encoded field of b bits, an escape cost of c bits, and a probability of failure in the encoded field of q, the average size of a pointer is $b+qc$. As N grows, b is constant, c grows, and if (6-2) holds, q approaches a constant q_0, $0 < q_0 < 1$. Of course, b should be chosen to make q small. Without (6-2), q will go to 1, the average size of a pointer will approach $b+c$, and we would do better to increase b or perhaps to abandon encoding altogether, particularly if $b+qc > \log_2 N$.

Of course the $p(i)$ depend on N, as equation (6-1) indicates. It will be convenient in what follows to express this dependence by writing $p(i)$ as a product of a normalizing term C, which is a function of N, and a function $f(i)$, which we assume to be independent of N. For example, for items distributed according to Zipf's law, $f(i)=1/i$ and $C=1/H_N$, where H_N is the N^{th} harmonic number. In general:

(6-3) $$p(i) = Cf(i)$$
$$C = \left(\sum_{i=1}^{N} f(i)\right)^{-1}$$

Now the question of whether a fixed width encoding will succeed--condition (6-2)--can be expressed as:

(6-4) $$\sum_{i=1}^{\infty} f(i) \quad \textit{converges.}$$

Starting from our empirical characterizations of the distributions of atom and list pointers, we will construct models of atom pointers and list pointers in the form of (6-3) and then evaluate condition (6-4) to see whether fixed-width encodings can maintain their efficiency as address space grows.

6.1 Atom pointers

In Section 1 (and in [Cla77]) we suggested that atom pointers are reasonably well modeled by Zipf's law, except perhaps for the most common few tens of atoms. Zipf's law states that the i^{th} most common object will occur with frequency proportional to $1/i$. If N is the number of atoms, then we can instantiate equations (6-3) for atom pointers as follows:

(6-5)
$$p(i) = C\,1/i$$
$$C = \left(\sum_{i=1}^{N} 1/i\right)^{-1}$$

Unfortunately the series

(6-6)
$$\sum_{i=1}^{\infty} 1/i$$

diverges, so if Zipf's law continues to hold as N grows, then the fraction of atom pointers encodable in any fixed number of bits will eventually decrease to zero. If, however, N is bounded then a fixed-width field of $\lceil \log_2 N \rceil$ bits will work regardless of how atom pointers are distributed. N would be bounded, for example, in a system whose atoms were the words of a natural language.

If N is not bounded, how might its growth be related to the growth of the address space? Two assumptions will help us answer this question. First, we assume that half of all *cars* will point to atoms (slightly more than the mean shown in Table I), and that no *cdrs* will (excepting NIL, as usual). If the address size is m bits and the number of atom pointers is A, then there will be 2^m list cells and $A = 2^{m-1}$. Second, we assume that the least common atom occurs once: $p(N)A = 1$. These assumptions allow us in principle to express N as a function of m. To simplify these calculations we will replace Zipf's law by the computationally more convenient *cumulative log law* [She78], which is very close to Zipf's law:

$$p(i) = \log_{N+1}[(i+1)/i]$$

(It is easy to show that the cumulative log law diverges in the sense of (6-4).) Applying the cumulative log law to our two assumptions yields Table VIII. As m grows from 16 to 32, Table VIII shows how many atoms there will be, what fraction of all atom pointers will be accounted for by the most common 1024 atoms, and how many pointers will be needed to account for 80 percent of all atom references. Although encoding efficiency falls, it does so quite slowly; even with 2^{24} list cells and more than 600,000 atoms, the top 1024 atoms will still attract more than half of all atom pointers.

TABLE VIII
Atom encoding as address space grows

address width (*m*)	number of atoms (*N*)	percentage of pointers to top 1024 atoms	number of atoms needed to get 80%
16	$3.96*10^3$	83.7	$7.54*10^2$
18	$1.37*10^4$	72.8	$2.04*10^3$
20	$4.86*10^4$	64.2	$5.61*10^3$
22	$1.74*10^5$	57.5	$1.56*10^4$
24	$6.27*10^5$	51.9	$4.34*10^4$
26	$2.29*10^6$	47.3	$1.22*10^5$
28	$8.40*10^6$	43.5	$3.46*10^5$
30	$3.08*10^7$	40.2	$9.79*10^5$
32	$1.17*10^8$	37.3	$2.85*10^6$

6.2 List pointers

In order to investigate the effects of address-space growth on list pointers, we rely on the hypothesis that static list pointer locality is largely a function of the time order of cell creation: cells close together in memory were very probably created close together in time. This property enables us to model list pointer locality by passing from the *addresses* of list cells to the *times* of their creation. Making this transformation allows us to model the growth of address space as the passage of time, and permits us to test proposed models against our programs.

We have considerable evidence for this hypothesis. The static measurements of [Cla77] showed that the so-called "smart *cons*" of Interlisp [Bob67, Tei74] has very little effect on pointer locality, and that a single free-list in address order is sufficient to account for this locality. Furthermore, in some experiments using two of the programs we directly compared address locality with creation-time locality and found that they were very close. An explanation for this is that even without a compacting garbage collector (Interlisp has none), free space for list cells tends to come in large blocks. A compacting garbage collector guarantees that the free storage list is a single huge block; addresses of cells created between compactions will correspond exactly to their creation times. Morris's compacting garbage collector [Mor78] will preserve this correspondence exactly; linearizing garbage collection will change it, but will improve pointer locality, as our data show. Therefore, ignoring the effects of garbage collection in what follows cannot weaken our convergence results.

We propose that the probability that a pointer in a cell created at time i points to a cell created at time j depends only on $|i-j|$. We assume this to be true after some large number, N, of cells has been created; but so as not to deal with cells created very near the beginning or very near the end, we assume $1 << i, j << N$. A pointer can point "forward" in time only if it is created by *rplaca* or *rplacd*; pointers "backward" in time can be made by either *cons* or *rplac* operations. Let $p(i)$ be the probability that a pointer points i units forward or backward in time. We assume that the probability has the same form regardless of which way the pointer was created.

An examination of our spatial locality data led us to the following model, a simple generalization of Zipf's law:

(6-7)
$$p(i) = C\, i^{-r}$$
$$C = \left(\sum_{i=1}^{N} i^{-r}\right)^{-1}$$

If $r=1$ then (6-7) is the Zipf model and does not converge. But if $r>1$ the series

$$\sum_{i=1}^{\infty} i^{-r}$$

does converge and a fixed-width encoding will therefore maintain its efficiency as N grows without bound.

Testing this model required that we gather new data on the distribution of pointers in list cells with respect to time of creation. We did this by patching the *cons* routine in Interlisp to put out a trace on a file of the addresses of cells as they were created during a typical run of two of the programs, NOAH and SPARSER. At the end of each run we dumped the contents of the list cells that had been created (17,800 cells in NOAH, 19,700 in SPARSER). We then translated the contents of each cell from virtual address to creation time and analyzed the results in terms of the fraction of cells that contained pointers at different distances (times of creation) away.

TABLE IX
Time locality in NOAH and SPARSER

	NOAH		SPARSER	
k	$B_k(car)$	$B_k(cdr)$	$B_k(car)$	$B_k(cdr)$
1	.537	.651	.501	.771
2	.084	.092	.053	.046
3	.044	.064	.049	.037
4	.020	.038	.027	.029
5	.028	.021	.033	.017
6	.020	.030	.030	.023
7	.011	.013	.034	.030
8	.014	.013	.026	.016
9	.023	.014	.015	.008
10	.022	.018	.026	.008
11	.028	.012	.036	.004
12	.036	.009	.040	.002
13	.056	.011	.065	.003
14	.067	.015	.046	.006

Table IX shows the result of collecting these statistics into buckets B_k containing pointer "distances" in the range $[2^{k-1}, 2^k)$. The numbers in Table IX are fractions of the total number of *car* or *cdr* list pointers that ended up in each bucket. Note that forward and backward time have been folded together, e.g., B_2 contains pointers to cells created 2 or 3 time units in the past or in the future. Note also that although a small number of pointers ended up in B_{15}, the experiments were not large enough to allow all possible values for pointers in B_{15}, so it is omitted from the table.

If these data were generated by model (6-7) with $r=1$, we would expect to see the values for B_k approach a constant as k increased (see, for example, Table III). For *car*, unfortunately, the situation is at least this bad: B_k seems to increase slightly as k goes from 8 or 9 to 14. Thus the model with $r>1$ cannot explain the *car* list pointer data. This, together with our conclusions about atom pointers, leads us to be pessimistic about the continuing efficiency of fixed-width encodings of *car*.

But for *cdr* there is some hope: B_k does appear to decrease with k, which is a necessary condition for (6-7) to hold. To examine this more closely we need to make some calculations. Our model predicts the following value for B_k:

(6-8) $$B_k = \sum_{2^{k-1} \leq i < 2^k} p(i) = C \sum_{2^{k-1} \leq i < 2^k} i^{-r}$$

For other than small values of k, this can be approximated by the continuous form:

$$\begin{aligned} B_k &\simeq C \int_{2^{k-1}}^{2^k-1} x^{-r}\,dx \\ &= C'[(2^k-1)^{1-r} - (2^{k-1})^{1-r}] \\ &\simeq C''\, 2^{k(1-r)} \end{aligned}$$

where C' and C'' are independent of k. Therefore:

$$\log_2 B_k \simeq C''' + k(1-r),$$

so our model predicts that $\log_2 B_k$ should be an approximately linear function of k with slope $(1-r)$ (for moderately large values of k, say above 3 or 4).

Figure 1 shows $\log_2 B_k$ versus k for the two sets of *cdr* data in Table IX. Also shown are plots of the exact values of $\log_2 B_k$ for model (6-7), computed according to (6-8) with $N=2^{14}$ and $r=1$, 1.2, and 1.3. There are several points to note. First, both sets of data decline quickly from $k=1$ to $k=2$, then very roughly linearly from $k=2$ to $k=14$. A slope of $(1-r)=-.3$ fits the SPARSER data adequately; $(1-r)=-.2$ does better for the NOAH data. Second, we can "adjust" model (6-7) without affecting its convergence by removing a finite amount of probability from its tail ($k>1$) and adding it to B_1. This would greatly improve the fit of the model to the data. Third, the sole property that we need for convergence is that the data always lie below some line with negative slope; the exact form of the model we have used--equations (6-7)--does not matter provided this property holds, and it seems to hold in Figure 1. In fact, r need not be constant. What is required is that it be bounded away from 1, for at $r=1$ the convergence disappears.

Thus if our assumptions remain valid, then we have demonstrated one plausible model of list-cell creation that roughly fits our *cdr* data and has the property that a fixed-width encoding of *cdr* will accomodate all but a bounded fraction of *cdr* list pointers as the total address space grows. Further testing with programs written for different Lisp implementations and different size address spaces is needed to increase our confidence in the model, and in the continued utility of compact encodings.

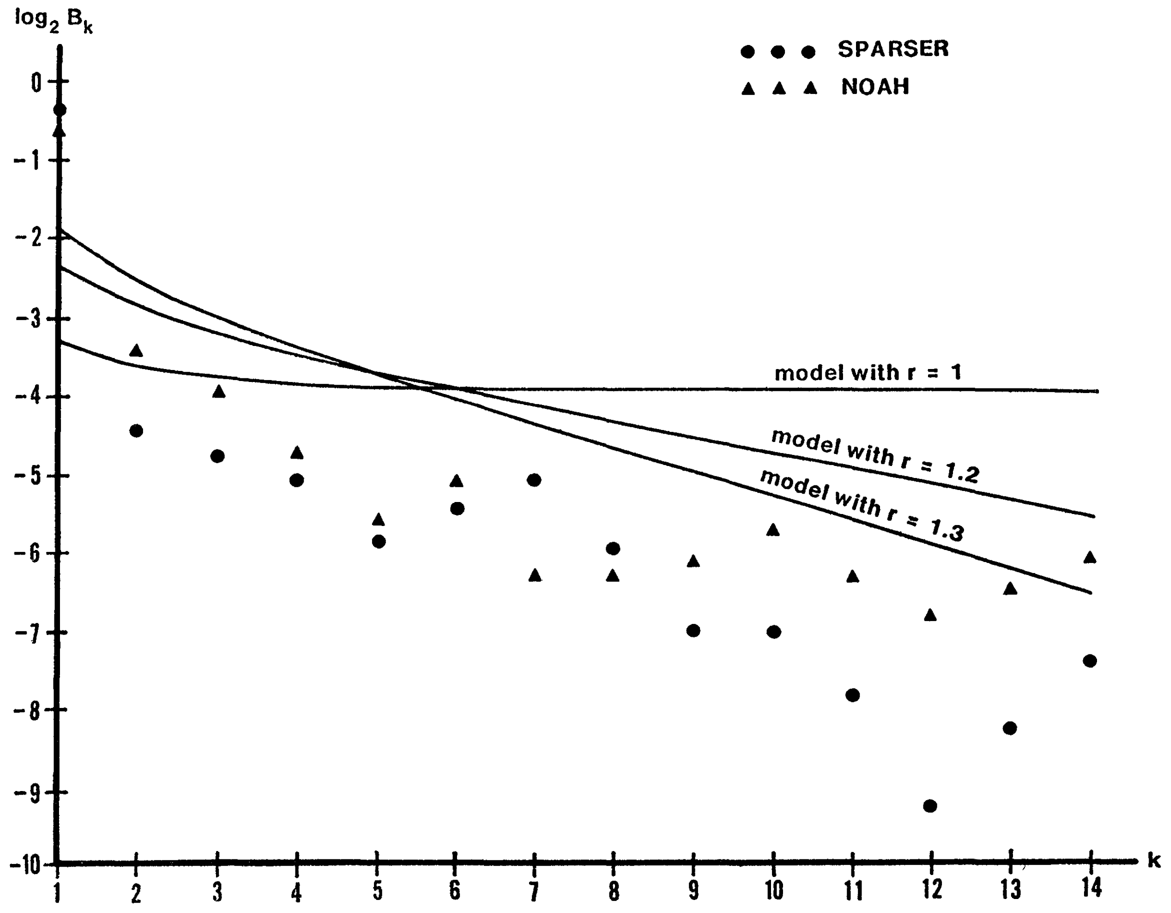

Figure 1

7. Conclusion

By taking advantage of the regularities in the use of list structure we can achieve a compact encoding of list cells. We have suggested ways to compress the average number of bits required for list pointers and for pointers to atoms. However, our models of the use of these pointer types suggest that as the total address space for list structure grows, only list pointers in *cdr* will be efficiently handled by a fixed-width encoding. Since *cdr* contains few pointers to atoms other than NIL, it is clear that there is a large benefit in encoding it. It is less clear that encoding *car* will be worthwhile. Encoding *cdr* only is the path taken by two groups who are building Lisp machines; our data and analysis indicates that they should prove quite successful in their uses of compact encodings of list cells.

Acknowledgments. We wish to thank a number of people who contributed to the development of these ideas and who helped us make and interpret our measurements. These include Lyn Bates, Mark Brown, Peter Deutsch, Therese Flaherty, Sam Fuller, Cordell Green, Leo Guibas, Ron Kaplan, Larry Masinter, Lyle Ramshaw, Earl Sacerdoti, and Beau Sheil. We also wish to thank Henry Baker and two other referees for detailed comments and suggestions which have helped us make our discussion more accurate and succinct.

References

[Bak78] Baker, H.G., Jr. List processing in real time on a serial computer. *Comm. ACM 21*, 4 (April 1978), 280-293.

[Bat75] Bates, M. The use of syntax in a speech understanding system. *IEEE Trans. on Acoustics, Speech, and Signal Processing 23*, 1 (Feb. 1975), 112-117.

[Baw77] Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D. LISP machine progress report. M.I.T. Artificial Intelligence Laboratory Memo No. 444, M.I.T., Cambridge, Mass., Aug. 1977.

[Bob67] Bobrow, D.G., and Murphy, D.L. The structure of a LISP system using two level storage. *Comm. ACM 10*, 3 (March 1967), 155-159.

[Bob72] Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM 15*, 3 (March 1972), 135-143.

[Bob75] Bobrow, D.G. A note on hash linking. *Comm. ACM 18*, 7 (July 1975), 413-415.

[Che70] Cheney, C.J. A nonrecursive list compacting algorithm. *Comm. ACM 13*, 11 (Nov. 1970), 677-678.

[Cla76a] Clark, D.W. An efficient list-moving algorithm using constant workspace. *Comm. ACM 19*, 6 (June 1976), 352-354.

[Cla76b] Clark, D.W. *List Structure: Measurements, Algorithms, and Encodings.* Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, August 1976.

[Cla77] Clark, D.W., and Green, C.C. An empirical study of list structure in Lisp. *Comm. ACM 20*, 2 (Feb. 1977), 78-87.

[Cla79] Clark, D.W. Measurements of dynamic list structure use in Lisp. *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 1 (Jan. 1979), 51-59.

[Deu73a] Deutsch, L.P. An interactive program verifier. Ph.D. thesis, Computer Science Dept., Univ. of California, Berkeley, May 1973.

[Deu73b] Deutsch, L.P. A LISP machine with very compact programs. *Third IJCAI*, Stanford, Ca., 1973, pp. 697-703.

[Deu76] Deutsch, L.P., and Bobrow, D.G. An efficient, incremental, automatic gabage collector. *Comm. ACM 19*, 9 (Sept. 1976), 522-526.

[Deu78] Deutsch, L.P. Experience with a microprogrammed Interlisp system. Proc. 11th Annual Microprogramming Workshop, Asilomar Conference Ground, Pacific Grove, Ca., Nov. 1978.

[Fen69] Fenichel, R.R., and Yochelson, J.C. A LISP garbage-collector for virtual-memory computer systems. *Comm. ACM 12*, 11 (Nov. 1969), 611-612.

[Gre74] Greenblatt, R. The LISP machine. M.I.T. Artificial Intelligence Laboratory Working Paper 79, M.I.T., Cambridge, Mass., Nov. 1974.

[Han69] Hansen, W.J. Compact list representation: definition, garbage collection, and system implementation. *Comm. ACM 12*, 9 (Sept. 1969), 499-507.

[Knu73] Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[McC62] McCarthy, J., *et al. LISP 1.5 Programmer's Manual.* M.I.T. Press, Cambridge, Mass., 1962.

[Min63] Minsky, M.L. A LISP garbage collector algorithm using serial secondary storage. Artificial Intelligence Project Memo No. 58 (rev.), M.I.T., Cambridge, Mass., Dec. 1963.

[Mor78] Morris, F.L. A time- and space-efficient garbage compaction algorithm. *Comm. ACM 21*, 8 (Aug. 1978), 662-665.

[Sac77] Sacerdoti, E.D. *A Structure for Plans and Behavior.* Elsevier North-Holland, New York, 1977.

[She78] Sheil, B.A. Median split trees: A fast lookup technique for frequently occurring keys. *Comm. ACM 21*, 11 (Nov. 1978), 947-958.

[Smi74] Smith, D.H., Masinter, L.M., and Sridharan, N.S. Heuristic DENDRAL: analysis of molecular structure. *Computer Representation and Manipulation of Chemical Information* (W.T. Wipke, S. Heller, R. Feldman, E. Hyde, eds.). John Wiley & Sons, Inc., 1974.

[Tei74] Teitelman, W. *INTERLISP Reference Manual.* Xerox Palo Alto Research Center, Palo Alto, Ca., 1974.

[Zip49] Zipf, G.K. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, Cambridge, Mass., 1949.

XEROX

XEROX

Compact Encodings of List Structure   By Daniel G. Bobrow and Douglas W. Clark

CSL-79-7