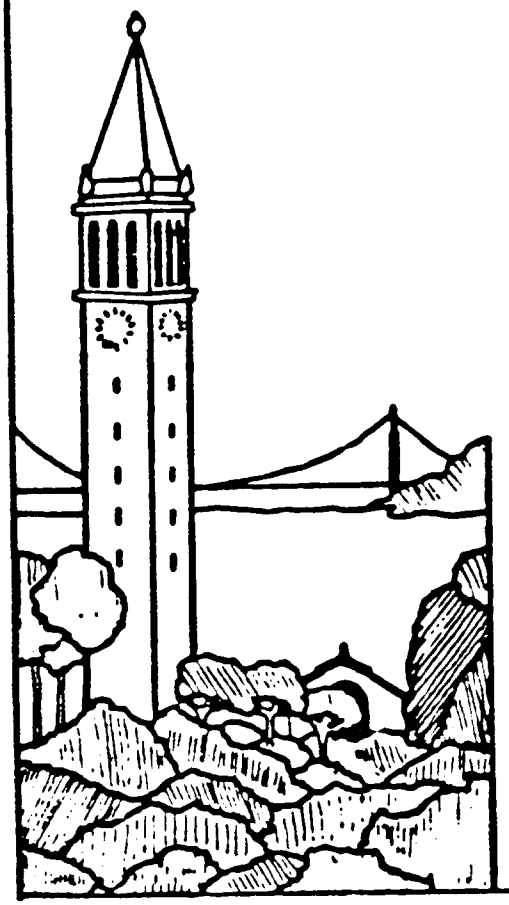
Data Structures and Destructive Assignment in Prolog

Patrick C. McGeer
Alvin M. Despain



Report No. UCB/CSD 87/356

July, 1987

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Data Structures and Destructive Assignment in Prolog

Patrick C. McGeer
Alvin M. Despain

Computer Science Division,
University of California,
Berkeley, CA 94720

1. Abstract

Two algorithms are considered for implementation in Prolog. It is shown that these algorithms cannot be implemented in pure Prolog and retain their nominal time complexity. It is shown that a generalization of Lisp's *rplaca/rplacd* construct suffices for these algorithms. We give a method for the implementation of the new construct, *rplacarg*, on a structure-copying Warren Abstract Machine and discuss implementation for a structure-sharing Warren Abstract Machine. We show how *rplacarg* may be implemented in CProlog using only pure Prolog constructs and the *var* construct. We show that *var* can be implemented in terms of the Prolog *not* primitive. We show that rplacarg can be used to implement multidimensional arrays efficiently in Prolog.

2. Introduction

The Aquarius Project [Despain85a] at Berkeley is developing high-performance computers [Despain85b] for the execution of Prolog. Part of the evaluation effort that we are making is to understand the advantages and disadvantages of Prolog for the implementation of programs to solve challenging problems in difficult domains of discourse. In particular, we have engaged in the design and implementation of a suite of Prolog CAD tools for VLSI design [Despain86] [Pincus86] [Bush87] [Cheng87] [McGeer87].

In the course of implementing a VLSI layout program in Prolog during the summer and fall of 1985, we experienced difficulties in implementing standard routing and transistor placement algorithms. After discussions with other groups that had used Prolog for Computer-Aided Design of Integrated Circuits programs, we concluded that the difficulties we experienced were common among Prolog CAD programmers. We investigated the nature and source of our difficulties, and concluded that the principal problem lay in Prolog's lack of a destructive assignment operator akin to Lisp's *rplaca* or *rplacd*. We then investigated the addition of such an operator to Prolog. This paper presents the results of that study.

This paper is organized into seven sections. Section II gives examples of the algorithms that we could not implement in nominal time in pure Prolog. Section III gives a general graph-theoretic argument to explain the difficulty of data-structure manipulation without destructive assignment. Section IV defines the destructive assignment operator *rplacarg* that we require and its operational and semantic characteristics. Section V describes an method for implementing the *rplacarg* in C-Prolog, or any implementation of Prolog that supports the *var* builtin. Section VI describes the features that must be added to a Warren Abstract Machine (WAM) [Warren83] to implement *rplacarg* for both the structure-sharing and structure-copying case. In particular, we show that a highly-efficient $O(1)$ *rplacarg* primitive may be added to our WAM-based Programmed Logic Machine (PLM). Section VII describes a multidimensional array implementation based on the *rplacarg* construct. In an appendix, we show that any implementation of Prolog that supports the *!, fail* implementation of negation supports *var* as well; hence we conclude that *rplacarg* is semantically implied by *cut* and *fail*.

3. II-Algorithms We Couldn't Implement Efficiently in Prolog

The central art of computer science is performing computations in the most time-efficient manner possible. Without efficiency concerns, all of computer science is trivia.

Concern for efficiency leads us to create data structures. Data structures are ways of storing intermediate results of computation, so that these computations need not be re-performed. Indeed, one might argue persuasively that all of computer science is the design of data structures that have the property that the amount of computation required to solve a given problem is minimized.

The core of our argument is that the implementation of some operations over some data structures is difficult and inefficient in Prolog, that these data structures are relatively familiar objects in some application domains, and that these difficulties arise precisely because of the applicative nature of Prolog. We have a general argument to explain this phenomenon, but our case can best be understood in light of a few examples.

We have not been able to implement the algorithms given below in nominal time in pure Prolog. By pure Prolog we mean Prolog without the well-known *assert/retract* primitives, which are known to be non-applicative (or, in the Prolog parlance, non-logical) or the *var* primitive, which we can show below is semantically equivalent to the non-applicative *rplacarg* primitive we advocate. Further, we can show that the well-known *cut, fail* construction for negation is equivalent to *var*, so we do not consider implementations using *cut, fail*.

3.1. Kernighan-Lin Min-Cut Algorithm

The Kernighan-Lin min-cut algorithm [Ullman82] is a greedy procedure to partition hypergraphs into two equal-sized sets so that the *cut* – the number of hyperedges that connect the two sets – is minimized. It has been shown that the min-cut problem for hypergraphs is NP-complete [Garey79]. However, the Kernighan-Lin algorithm is an excellent approximation procedure.

The Kernighan-Lin algorithm begins with the nodes of the graph partitioned arbitrarily into the two sets, called *left* and *right*. On each iteration, a pair of nodes *(l, r)* is selected for interchange; the pair

selected is that creating the greatest decrease or smallest increase in the *cut*. The pair are not immediately interchanged; but are merely marked as selected, treated as interchanged, and removed from *left* and *right*. When *left* and *right* are empty (there are no more unselected nodes), the total summed cost of the interchanges are computed in order. The largest negative total is taken, if there is any, those paris of nodes are interchanged, and the selection process begins on the new *left* and *right*; if no negative total is found, the algorithm terminates.

The minimum requirement of this algorithm is that the cost of each interchange be rapidly computed. This in turn implies that each hyperedge have a pointer to each node upon which it is incident. Similarly, once a node is selected, it must be marked as selected; the selection, or not, of a node affects future cost computations on hyperdges incident upon that node. If marking a node as selected involves regenerating the node (as it does if neither *var* nor some form of destructive assignment is used), each hyperedge incident upon that node must be regenerated. There are potentially 2^{n-1} such hyperedges on an n node hypergraph, and hence this is quite an expensive operation. Similarly, when the nodes are interchanged, if an interchange requires regeneration of each node, then every hyperedge must be regenerated. There are at most 2^n hyperedges on an n-node hypergraph.

We provide our Prolog implementation in an appendix, using our *rplacarg* primitive, to be discussed in section IV.

3.2. $O(n)$ Average-Case Sorting

Jon Bentley [Bentley84] has posed a puzzle in sorting. Given two integers N,M, with $N<M$, generate N distinct random numbers in the range $[0,M]$ and print them out, sorted, in *average-case* time $O(N)$.

Clearly this problem cannot be solved in worst-case time better than $O(N\log N)$. However, Knuth [Knuth86] has posted a solution to this puzzle with average-case behavior $O(N)$, and worst-case behavior $O(N^2)$.

The core of Knuth's method is the use of a hash table of size N, which is simply a vector. Implementation of vectors in Prolog has proven quite troublesome, and there have been a number of proposals. In section VII we show that the central difficulty in the implementation of vectors is the avoidance of copying the entire vector when a single argument is set. This is precisely the problem that we are trying to solve, and so it is unsurprising that our proposal here makes the implementation of arrays quite easy. For the moment, we just note that we can use the builtin *functor* to get storage, and assume that in any rational Prolog implementation *arg* is $O(1)$, and hence can be used for indexing.

Knuth uses a monotone-increasing function to hash each random number into the hash table, and uses an insertion-sort to resolve collisions. Clearly the hash table remains sorted; if there are a very small number of collisions, then time of the algorithm is $O(N)$; in fact, the probability of collision is very small. It is possible, however, for all numbers to hash to the same bucket, in which case Knuth's time is $O(N^2)$.

We have devised a Prolog algorithm, using *var*,which matches Knuth's performance. Instead of maintaining only one item per bucket and using an insertion sort to avoid collisions, we maintain a bucket as a list and sort the list on output. In order to avoid having to replace the entire array, we maintain an unbound cdr on each bucket. The algorithm appears here.[1]

```
:- consult(rand_int).

table(Seed):- M = 24, N = 96, M2 is 2*M, functor(S,s,M2),     % Initialize.
        fill_table(M,N,S,Seed,M),
        S =.. [_|SS],
        print_sort(SS),nl,
      print(' DONE ! '),nl.

/* fill the hash table */

fill_table(M,N,S,Seed,0).
fill_table(M,N,S,Seed,I) :-
        rand_int(Seed,Snew,1,N,T),
        H is 1+2*M*(T-1)//N,
        arg(H, S, V), insert(T,V,I,J),
```

[1] If we use a random number generator with period $> M$, then all generated samples are distinct and we need not check for duplicates. If there are no duplicates, then the only entry in a bucket which might unify to a list containing a new entry is the unbound *cdr*, and hence we do not need *var*

```
fill_table(M,N,S,Snew,J).

/* Insert an element into the table, maintaining the unbound cdr */

insert(X, Y, I, J) :- var(Y), !, Y = [X|_], J is I-1.      % Insert element.
insert(X, [X|_], _, _) :- !.                               % eliminate dups
insert(X,[H|T],I,J) :- insert(X,T,I,J).                    % Skip down list.

/* If a bucket is empty, it is unbound, and hence unifies to kruft (or, for
that matter, any atom). If it is nonempty, it is a list, and hence won't
unify */

print_sort([]).
print_sort([kruft|T]) :- !,print_sort(T).          % Strip empty lists
print_sort([H|T]) :- sort(H,C),lprint(C),pr_st(T). % Print 1th bucket

pr_st([])    :- print('.'), nl.                    % Terminate the printout
pr_st([kruft|T]) :- !, pr_st(T).                   % Strip out empty lists
pr_st([H|T]) :- print(','), sort(H,C),             % Sort the bucket
                lprint(C) , pr_st(T).              % Print the bucket

lprint([H])  :- print(H).                          % Print last item
lprint([H|T]) :- print(H),print(','),lprint(T).    % Print list element
```

Now, in the average case there are a small number of collisions, and hence our algorithm, like Knuth's, is $O(N)$. In the worst case, where every number hashes to the same bucket, the cost of sorting the bucket is $O(N\log N)$ but the worst-case time is determined by the cost of adding items to the end of the list. This, of course, is $\sum_{i=1}^{N} i = O(N^2)$

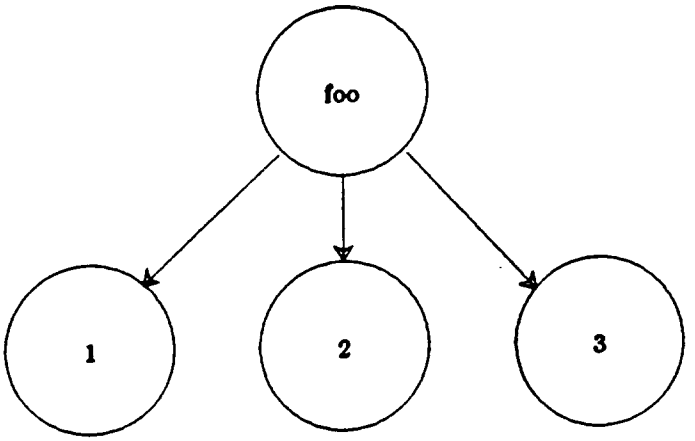
Of course we could do somewhat better than this if either we could prepend to the list or maintain a form of balanced tree rather than a list with an unbound cdr. Unfortunately, either balancing a tree or prepending items to the list involves generating a new tree or list, and thus changing the appropriate entry in the hash table. But changing the appropriate entry in the hash table *without* copying the entire hash table (an $O(N)$ cost for every new random number, giving us a worst-case time of $O(N^2)$) requires some form of destructive assignment. It is quite easy to see that if some form of destructive assignment is employed, the worst-case time of the algorithm goes to $O(N\log N)$, which is nominal for this problem.

If we did not use *var*, then this hash-table algorithm would require copying the hash table in the event of a collision. This gives a worst-case time complexity of $O(N^2)$, which is the same as the

implementation using *var*. The space complexity of the algorithm using *var* is $O(N)$, however, and the space complexity without var is $O(N^2)$).

4. III -- A Graph-Theoretic View

In order to understand why the above examples are difficult to solve efficiently in a purely applicative manner, we need an abstract view of the data structures created and used by programs. We picture a program's data structure as a dynamic graph, whose nodes are the records used to instantiate the structure and the atoms and constants in use by the program, and whose edges represent pointers to substructures. For example, the data structure *foo*(1,2,3) is represented as the graph:



Atoms and constants are nodes of outdegree 0. Unbound variables have *indegree*=*outdegree*=1, and their sole edge points to themselves.

Only nodes with *indegree*=0 are accessible at the top level of the program; we call these nodes the *roots* of the program's data structure.

In general, we assume that it costs no more to generate a node and set its outbound edges than it does to visit it. Hence, we are concerned with the number of visits that we must make to a node.

We may create or destroy nodes, but the nodes themselves hold no value; here we are unconcerned with the internal value of a node (in the case of an internal node, its type -- eg the name of its functor; in the case of an atom or constant, its name or value). All we are concerned with here is reassigning the edges of the graph. If a constant field changes in a structure, we depict this by changing the edge

representing the field from the old constant node to the new.

4.1. Principles of Modification

Now that we have a graph-theoretic model of data structures, we can turn our attention to the principles that we wish a modification operation to have. First, we have a *Principle of Consistency*: if a node in the graph is modified, then either the modification must be invisible to all ancestors in the graph *or* those ancestors must be modified to be ancestors of the modified node. Second, we have a *Principle of Atomic Modification*: if a clause C modifies a node N, it should not have to modify any other node in the graph solely to maintain the principle of consistency.

The principle of consistency is clearly just correctness in a more specific guise. The principle of atomic modification is a consequence of various principles of structured programming and language design, principally abstraction and information hiding.[2] There is no reason to believe that an arbitrary program data structure graph is *homogenous* (in other words, all nodes are of the same type). Clearly, then, if a clause is forced to modify an arbitrary number of nodes in the graph, it is potentially forced to modify nodes of any type. Clearly this contradicts any reasonable definition of modularity or information hiding. Indeed, one can argue that an important consequence of the structured programming revolution is the notion that a procedure should operate on only a finite number of types, *independent of the number of types defined by the entire program.*

A weaker form of the principle of atomic modification may be derived on complexity grounds. In general, the number of nodes in a program's data structure graph at any time is polynomial in the size of the input. We can certainly devise programs in which the number of reassignments of graph edges is of the same order as the complexity of the program. Hence if the number of modifications a clause must make in order to maintain the principle of consistency is not bounded above by some integer $k \geq 0$ independent of the size of the input, then the complexity of the program will not be nominal. From these considerations we derive the *Principle of Bounded Modification*:if a clause C modifies a node N, it should

[2] For a thorough discussion of such principles, see, eg, [MacLennan83]

not have to modify more than k other nodes in the graph, for k an integer ≥ 0, independent of the size of the input.

It is very unlikely that any modification discipline that guarantees consistency over a range of programs and data structures may violate the principle of atomic modification and nevertheless uniformly respect the principle of bounded modification. Hence it seems very likely that these two principles are in fact equivalent.

The only form of assignment permitted by Prolog is that an unbound variable's sole edge may be assigned to point to anything (or, equivalently, the variable's node may be replaced in the graph by any subgraph). A more general form of assignment, not permitted by pure Prolog, permits edges to be reassigned once assigned.

Prolog's form of assignment raises the possibility of conflict between the principle of consistency and the principle of atomic modification. If a node N is to be modified in Prolog, then the node must be regenerated, and a new node N' created. All ancestor nodes to N must be modified to point to N' in order to maintain the principle of consistency; the principle of atomic modification forbids the procedure that generates N' from modifying the ancestor nodes.

We immediately observe that there is no conflict between the two principles under Prolog's form of assignment if the program's data structure graph is a forest of trees. Let N be modified by clause C to N'. Now, either N is a root or it is not. If N is a root, then it has no ancestors and hence no other nodes need be regenerated in order to maintain the principle of consistency, and hence the principle of atomic modification is not violated. If N is not a root, then it has a set of ancestors say $N_1, \ldots, N_k$, and the set has been traversed by a set of clauses $C_1, \ldots, C_k$, where clause C_i traversed node N_i, N_i is the parent of N_{i+1} in the program's data structure graph and C_i is the parent of C_{i+1} in the program's proof tree (or, if you prefer, calling tree). Hence C_i may generate N_i', where N_i' is identical to N_i save that it is the parent of N_{i+1}' rather than N_{i+1}. Since each clause modifies one and only one node in the data structure graph, the principle of atomic modification is upheld.

If the program's data structure graph contains networks or more general graphs, then the principles are in conflict indeed. The difficulty is that node N in a network has several parents, only one of which is known to be an argument to a clause in the program's proof tree. In the case of a tree above, the graph could be easily modified since the set of nodes which had to be regenerated were visited in the natural course of satisfying the program's proof tree. In the case of a network, this is not the case. The principle of consistency cannot be maintained simply by upward traversal of the program's current proof tree. Rather, the set of parents must be found by explicit traversal of the program's data structure graph and directly modified. Since this procedure is recursive, *potentially the program's entire data structure graph must be immediately regenerated*, which is a clear, and serious, violation of the principle of atomic modification. It is also, in general, a violation of the principle of bounded modification.

Prolog programmers therefore have three choices. First, we may use only trees or simplifications of trees (such as lists, simply a special case of a binary tree); second, we may violate the principle of atomic modification, which in practice makes many programs expensive and difficult to write; or we may choose to embrace a more general form of modification.

5. IV -- Requirements for a General Form of Modification

The preceding argument shows the general requirements for a general form of modification. First, any such operation must follow the two principles laid down in the preceding section. Second, such an operation permit atomic traversal of any edge in the program data structure graph. Third, values of variables and structure components form part of the state of the program at any time; backtracking restores program state, and hence must restore variable values. Therefore assignments must be undone automatically on backtrack. Fourth, fully general assignment such as Lisp's *setq* is not required; all that is required is some method of manipulating arguments of structures atomically.

5.1. Methods of Representation of Data Structures.

For obvious reasons, the method of modifying data structures is bound up in their representation. We examine three options:

5.1.1. Use of the Prolog Database, and Modifications using *Assert/Retract*

This has been a popular choice among Prolog CAD programmers [Hill84], but we find it unsatisfactory for several reasons. First, we find that one of the strengths of Prolog is its ability to equate several variables without assigning any of them to values; an assignment to any one therefore assigns to them all. *Assert* destroys such links between logical variables. Second, links between nodes in the data structure graph must be maintained through some form of keys, and the Prolog database search mechanism employed to search for the successor nodes. This search may *appear* to be $O(1)$ to the programmer, but an actual $O(1)$ search on a procedure that changes during the course of a program's execution requires an adaptive hashing scheme beyond that employed by most Prolog execution environments. On another level, the use of such keys is really a form of explicit pointers, and one of the major motivations for symbolic programming languages has historically been the desire to avoid explicit pointers. Third and most important, such modifications are not undone on backtrack, which we (and most Prolog programmers) find unacceptable.

5.1.2. Use of Secondary Storage Structures with Explicit Keys

In this method, rather than storing the actual pointers to successor nodes, nodes store keys and search a secondary structure which *may* be easily modified for the value.[8] We have two objections to this. First, structures which may be modified easily are trees, and hence the cost of any modification is bounded below by $\log n$, and above by n. Second, the objection to explicit pointers cited above applies here. Third, additional storage structures unnecessarily complicate the code.

5.1.3. Use of Implicit Pointers and an Explicit Assignment Mechanism, *rplacarg*

We prefer to manipulate pointers implicitly, in the manner of classic Prolog and Lisp programs. In order to do this, we need an explicit mechanism to reset pointers.

---

[8] see, eg, [Bratko86]

The mechanism we choose is a generalization of Lisp's *rplaca* and *rplacd* mechanisms. Our mechanism, *rplacarg(Term, ArgNum, Value)*, sets the argument *ArgNum* of term *Term* to *Value*. No unification is done on *Term*, other than to determine that it has at least *ArgNum* arguments, and to determine the address of *ArgNum*. The value *Value* is then written into the appropriate location, and the old value and the address trailed.

Notice that *rplacarg* when called on a list with *ArgNum*=2 is equivalent to *rplacd*; when *ArgNum*=1 it is equivalent to *rplaca*.

When *rplacarg* is used to manipulate edges, both the principles enumerated in the previous section are respected; consistency is maintained, since the assignment is transparent to all other nodes in the graph, and atomic modification is maintained since only one memory location (and hence only one node) is affected. Further, structures are represented naturally, without explicit indices; no secondary data structures are required, and hence pointer traversal is $O(1)$.

6. V -- Implementation of Rplacarg in Quasi-Pure Prolog

Quasi-pure Prolog is Prolog code that does not use *assert*, *retract* or *write*, but that does use *cut*, *fail* and other built-in meta-logical primitives such as *var*. In this section, we demonstrate an implementation of *rplacarg* using the *var* primitive.

Conceptually, what we want to do here is permit programs written in Prolog to behave *as if* Prolog was a language that permitted multiple assignments, when in fact it permits only a single assignment. In order to do this, we must store rather more than the value of some component of a structure in its slot; we must store a data structure, containing at least the current value of the slot and an unbound variable; the unbound variable is reserved to be bound to future values of the component. Both an inductive view of this requirement and the need to save old values against backtracking indicate that all old values of the component mus tbe stored in this structure.

The simplest structure which performs these tasks for us is a list, whose last element is an unbound variable and whose remaining elements are past values of the component, in order; the first element of the

list is the first value of the component, and the last (but one) is the current value of the component. Accessing the current value involves traversing the list until the last bound element is reached, and returning that value; setting the current value involves traversing the list until the last element is found, and then binding that element to a list consisting of the current value followed by an unbound variable.

To avoid semantic confusion when either unbound variables or lists become values of the component, we use an equivalent data structure, which we call a *valStruct*; a *valStruct* has two components, *value* and *futureValues*. The equivalence of a *valStruct* to a list is easily seen if it is remembered that the Prolog list operator is merely syntactic sugar for the binary operator . , which was the list operator in early Prolog implementations.

We formalize these notions in two procedures: *accessVal* and *setVal*. *accessVal* accesses the current value of such a nested *valStruct*; *setVal* sets a nested *valStruct* to a new value.

```
accessVal(valStruct(X, U), X) :-
   var(U).

accessVal(valStruct(_, Y), X) :-
   accessVal(Y, X).

setVal(U, X) :-
   var(U),
   U == valStruct(X, _).

setVal(valStruct(_, Y), X) :-
   setVal(Y, X).
```

Once this construct is adopted it is relatively easy to write *rplacarg*:

```
rplacarg(Term, ArgNum, Value) :-
   arg(Term, ArgNum, Arg),
   setVal(Arg, Value).
```

It is relatively easy to see that this implementation of rplacarg meets our criteria; in particular, old values are restored on backtrack. It does, however, create three problems:

(1) Since components of data structures no longer contain only the value of the component, programs

cannot use the unification mechanism of Prolog to examine structures directly; rather, they must use the analog to *rplacarg*:

```
accessarg(Term, ArgNum, Value) :-
    arg(Term, ArgNum, Arg),
    accessVal(Arg, Value).
```

This is not a major problem for us, since we prefer access procedures and type definition code to unification in any case: it makes modification of the definition of data structures easier during program development. Many Prolog programmers, however, find the unification mechanism extremely helpful.

(2) Access times can no longer be bounded by $O(1)$; rather, each access (or set) consumes time proportional to the number of times a component is set during the course of an algorithm; of course, this number may be proportional to the time complexity of the algorithm, though in general it is $O(1)$. Hence this implementation can in a pathological case square the running time of an algorithm.

(3) This method stores all old values of every component, which is extremely space-inefficient. We shall show below that an old value need be stored only in a subset of the cases where the address of the component would need to be stored if bound as an unbound variable. As shown by Warren and others[Tick86] experimentally, this is only a small percentage of the cases. Hence most of the storage used by this algorithm is garbage, and, worse, garbage that cannot be collected by most garbage collection algorithms.

In sum, this method permits the development of programs using networked data structures in current Prolog implementations; it also serves to show that *rplacarg* is no worse a corruption of pure Prolog than *var*.

7. VI -- Implementation in a Warren Abstract Machine

The Warren Abstract Machine [WAM] is a three-stack architecture for the execution of Prolog. Virtually every Prolog implementation assumes some variant of the WAM, or implements one, all the way from interpreters through dedicated hardware.

In most respects, the WAM is a conventional stack-based Von Neumann architecture. The WAM's *local stack* resembles the stack on most conventional machines. The stack contains two types of data structures, *environments* (analogous to and closely resembling stack frames in conventional architectures), and *choice points*. These are required to support the non-determinacy of Prolog programs. They save the register values and form a "cap" on the stack which cannot be removed until this choice point is either exercised or removed by a *cut*. The second stack, the *heap*, is precisely analogous to the heap in Algol-60, and performs the same function. The third stack, the *trail*, has no analogue in non-WAM machines. Its purpose is to save the addresses of variables which have been set, so that these variables may be unset upon backtrack.

Clearly not all values need be reset upon backtrack. In particular, variable locations above the top of the heap when the last choice point was laid down will disappear on backtrack, and hence need not be reset; similarly, variables above the top choice point on the stack need not be reset. WAM architectures perform both these optimizations.

7.1. Structure-Copying Machines

On structure-copying machines, *rplacarg* is an extremely simple operation to implement. In such machines, an n-field structure takes up n+1 consecutive locations on the heap. The first location contains the functor and arity information; the remaining n contain the n arguments, in order. Hence implementing *rplacarg* requires only finding the base address of the structure on the heap, indexing to the argument to be written, and writing it directly; no unification is involved.

Of course, the *rplacarg* operation must be undone on backtrack, so if the location written must be trailed as if written originally, and its original value trailed with it. The usual optimizations apply; if this location will disappear in any case on the next backtrack, then the trailing need not be done.

The need to trail values as well as locations means that trail entries must become a pair rather than a single entry. Strictly speaking, trail entries need only be a pair if the previous entry was a value, rather than the special value *unbound*; however, we suspect that the penalty for making each entry on the trail a pair rather than discriminating on this basis is too small to warrant the additional implementation

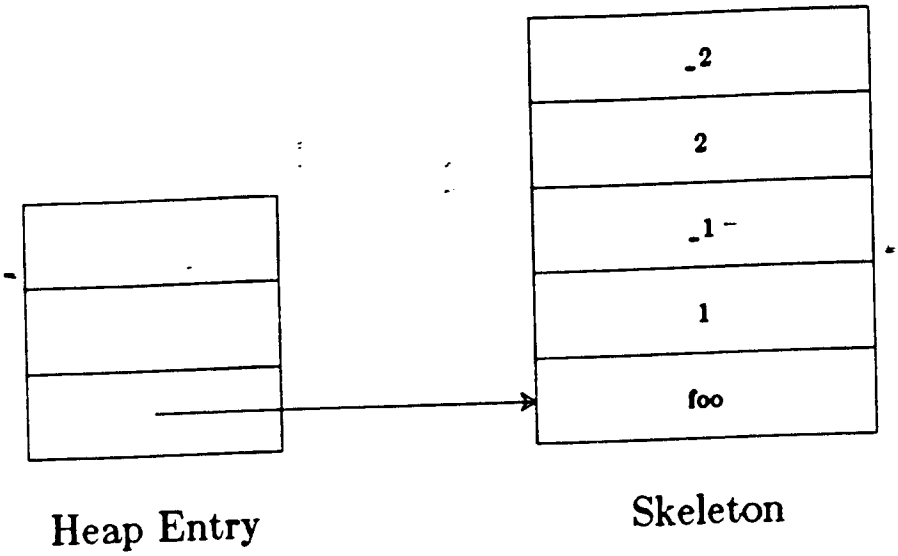
complexity.

On a side note, this implementation adds some garbage to the trail. Suppose some location *k* is written twice after some choice point has been laid down and before the next one is laid down; *k* will be written twice on the trail, and on backtrack will have two values restored, only the second of which is at all relevant to future computation. Touati[Touati86], however, has demonstrated that it is a small matter to garbage-collect the trail.

7.2. Structure-Sharing Machines

Structure-sharing implementations of the WAM do not directly represent a structure on the heap in the straightforward manner of structure-copying implementations. Rather, a structure is represented on the heap by k+1 consecutive locations, where k is the number of variables appearing in the *skeleton* of the structure, that is, the instance of the structure appearing at some location in the program. This practice saves some heap space when constants appear in structures in the program, since the structures' constant arguments are not copied onto the heap.

In a structure-sharing implementation[Warren77], the first of the k+1 heap locations contains a pointer to the skeleton in code space, and the remaining k arguments provide values of the variables referenced in the skeleton. For example, the structure *foo(1, X, 2, Y)* would be represented as:



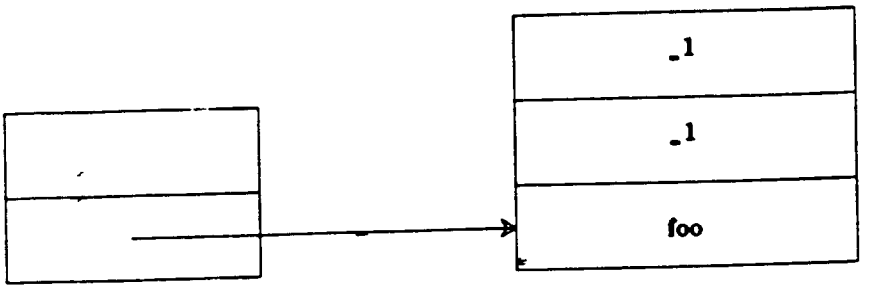
foo(1, X, 2, Y): Structure-Sharing Implementation

_n in the diagram refers to an offset of n locations from the base of the heap entry.

Space-saving is achieved since the skeleton, which is at least as large as the heap entry, appears only once, while the heap entry is created as often as the structure based on this skeleton is instantiated. In a structure-copying implementation, the heap entry is the same size as the skeleton.

Unification is more complex in a structure-sharing environment, and for obvious reasons. *rplacarg* is more complex in a structure-sharing environment as well. First, the skeleton must be referenced to determine which heap location must be written. It may be that the appropriate argument in a structure-sharing environment is *not* a heap location (for example, arguments 1 or 3 in the above example), in which case the replacement should not take place, since the replacement would occur in every instance of this skeleton on the heap; clearly not what is desired. *rplacarg* must fail, or, better, signal an error.

More subtle bugs may occur in a structure sharing environment. Consider the skeleton *foo(X, X)*. The diagram appears below



Heap Entry Skeleton

foo(X, X): Structure-Sharing Implementation

Now, suppose it was desired to replace the first argument of some instance of this structure: if it had been 1, suppose it was written to 2. *rplacarg* would access the first argument in the skeleton, find that it was a heap offset (offset of 1), would find the location in the heap and then write it with the value 2. Subsequent accesses to the second argument would also find that its value was 2, because *the first and second argument reference the same memory location.*

Note, incidentally, that this bug afflicts the implementation of *rplacarg* given in the previous section. This has not affected the programs we have written using this mechanism (including a 3000-line circuit layout package) since our data structure definition packages do not create skeletons containing either constants or repeated variables.

The solution to this bug is to attach a non-writeable protection flag (it need only be a single bit) to each argument of a skeleton in Prolog. If the flag is FALSE, then the argument can be written; if it is TRUE, then the argument cannot be written. This is not a difficult task, since the writeability of any argument is determined when the skeleton is created, and is quite easy to determine: the only writeable arguments are those which are variables which only appear once in the skeleton.

8. VII – A Note Concerning Arrays

A number of array implementations have been proposed for Prolog in recent years.[4] Most such

[4] See, eg [Cohen84], [Touati86]

implementations use the *assert/retract* primitives of Prolog, or propose new data areas to contain the array, or some combination of these effects.

If *rplacarg* is admitted, arrays fall quite naturally into standard Prolog as just another form of structure. The principle difficulty that people have in forming arrays is that the necessary relationship between the addresses of the various elements means that the graph of the array data structure is, in some sense, complete; each element of the array has an implicit pointer to every other element of the array. Hence any modification of any element of an array under a purely applicative model of computation requires copying the entire array, as discussed in section III. Once the applicative model is disposed of – and in section IV we see it does not apply to Prolog, in any case – array implementation becomes quite easy.

An array is merely a data structure with two fields – a *dope vector*, which describes how a given element may be found, and a one-dimensional vector of storage which we call a *hunk*, which contains the elements. Now, under any reasonable Prolog implementation data structures will be stored contiguously in memory, so we use the built-in CProlog primitive *functor*, which creates a term of arbitrary size.

We give the code to make and access arrays here. Note that arrays here are structures of four components; the fields *Dimension* and *DimensionVector* are included merely for error-checking.

The code is relatively straightforward, and should be easy to follow. *makeArray(DimensionVector, Array)* makes a multi-dimensional array of size indicated by *DimensionVector*, which should be a list of positive integers; *accessElement(Array, IndexVector, Value)* returns the appropriate element of *Array* in *Value*; of course, *IndexVector* should be a list of positive integers of the appropriate size of appropriate values. *setElement(Array, IndexVector, Value)* sets the appropriate element of *Array* to *Value*. The other routines appearing here are required for support.

The actual implementation of arrays in CProlog 1.5 was a little more complex than this, since CProlog only permits terms of size 100; readers who wish the array package should write the authors. The point of this section is merely to demonstrate that, once *rplacarg* is admitted in Prolog, then the implementation of arrays is quite natural in Prolog, and requires no other extension to the language.

```
/* Code to make an array.  the dimension and Dimension vectors are
unnecessary; in fact, dimension is so far unused.  Dimension vectors are
good for error checking during access... */

makeArray(DimensionVector, array(Dimension, DimensionVector, DopeVector, Elements)) :-
    makeDopeVector(DimensionVector,Dimension,Size,DopeVector),
    allocateStorage(Size, Elements).

/*
Make the dope vector for the array; the idea is to make address calculation
simple...ie., if the index vector is i[1],i[2],i[3] and the dope vector is
d[1], d[2], d[3], the address is:
i[1]*d[1] + i[2]*d[2] + i[3]*d[3]
*/

makeDopeVector([],0,1,[]) :- !.     /* Size of 1 is a hack for the usual case..*/

makeDopeVector([Dim|_], _, _, _) :-
    Dim <= 0,
    write('Error -- size <= 0 in a dimension of this array'), nl, !,
    fail.

makeDopeVector([Dim|Rest],Dimension,Size,[Size1|DopeVect]) :-
    makeDopeVector(Rest,Dim1,Size1,DopeVect),
    Size is Dim * Size1,
    Dimension is Dim1 + 1, !.

allocateStorage(N,Storage) :-
    functor(Storage,hunk,N).

/* Dig the value of an element out */

accessElement(array(Dimension, DimensionVector, DopeVector, Elements),IndexVector,Val) :-
    calculateArg(DopeVect,DimensionVector,IndexVector,Arg2),
    Arg is Arg2 + 1,
    accessarg(Arg,Elements,Val).

/* Set an Element */

setElement(array(Dimension, DimensionVector, DopeVector, Elements),IndexVector,Val) :-
    calculateArg(DopeVect,DimensionVector,IndexVector,Arg2),
    Arg is Arg2 + 1,
    rplacarg(Arg,Elements,Val).

/* Calculate the arg (offset) of an element from its dope vector.  The Index
Vector is given merely for error-checking */

calculateArg([],[],[],0) :- !.

calculateArg([],[],_,_) :- !,
    write('error -- too many dimensions in access'), nl, !,
    fail.
```

```
var(X) :- not(not_a(X)), not(not_b(X)).

not(X) :- X, !, fail.
not(_).
```

Of course, this variant of negation is somewhat controversial in the Prolog community, espescially when it is applied to non-ground terms (as it is here)[Flanagan86]. However, we suspect that we could write a such a *var* procedure in most reasonable forms of negation; moreover, since we immediately backtrack over the bindings we make, we are not troubled by inconsistent bindings.

11. Appendix -- Implementation of Min-Cut Algorithm in Prolog Following is the code for the min-cut algorithm, implemented using *rplacarg* in Prolog. We use *setField* and *accessField* as symbolic synonyms for *rplacarg* and *accessarg*.

```
% min-cut algorithm. Given a partition of the graph, find a new partition
% so that the cut is minimized.

min_cut(U, V, NewU, NewV) :-
   turn_off_selections(U),
   turn_off_selections(V),
   min_cut_loop(U, V, Selections),
   min_cut_movelist(Selections, Moves),
   min_cut_check(Moves, U, V, NewU, NewV).

turn_off_selections([]) :- !.

turn_off_selections([Block|Blocks]) :-
   setField(Block, selected, false),
   turn_off_selections(Blocks).

% End of algorithm, or try again? If Moves are [], can't improve placement.

min_cut_check([], U, V, U, V) :- !.

min_cut_check(Moves, U, V, NewU, NewV) :-
   make_moves(Moves, U, V, NextU, NextV),
   min_cut(NextU, NextV, NewU, NewV).

% make moves. Looks weird, but I swear it's faster this way O(2n) instead
% of O(n^2).

make_moves([], U, V, NewU, NewV) :-
   concat(U, V, L),
   partitionOntoSides(L, NewU, NewV).
```

```
make_moves([cost(U0, V0, _)| Moves], U, V, NewU, NewV) :-
   setField(U0, side, right),
   setField(V0, side, left),
   make_moves(Moves, U, V, NewU, NewV).

partitionOntoSides([], [], []) :- !.

partitionOntoSides([Block|Blocks], [Block|Lefts], Rights) :-
   accessField(Block, side, left),
   !,
   partitionOntoSides(Blocks, Lefts, Rights).

partitionOntoSides([Block|Blocks], Lefts, [Block|Rights]) :-
   partitionOntoSides(Blocks, Lefts, Rights).

% main loop. Trivial Cases.

min_cut_loop([], _, []) :- !.

min_cut_loop(_, [], []) :- !.

min_cut_loop(U, V, [Selection|Selections]) :-
   infinity(Inf),
   min_cut_select(U, V, cost(_, _, Inf), Selection),
   Selection = cost(U1, V1, Cost),
   (Cost = Inf -> write('Selection unbound!'), nl, break; true),
   setField(U1, selected, true),
   setField(V1, selected, true),
   delete(U, U1, Up),
   delete(V, V1, Vp),
   min_cut_loop(Up, Vp, Selections).


% trim the selections made by min_cut_loop down to a movelist.

min_cut_movelist(Selections, RealSelections) :-
   find_min_point(Selections, 0, 0, 0, 0, N),
   trim_selections(Selections, N, RealSelections).

% find the point where the sum is minimum.

find_min_point([], _, _, _, N, N) :- !.

find_min_point([cost(_, _, Cost)|Sels], CostIn, CurMin, LastPt, MinPt, N) :-
   ThisCost is CostIn + Cost,
   ThisPt is LastPt + 1,
   (ThisCost < CurMin ->
        find_min_point(Sels, ThisCost, ThisCost, ThisPt, ThisPt, N)
        ;
        find_min_point(Sels, ThisCost, CurMin, ThisPt, MinPt, N)
   ).

% Now trim selections, guided by N.
```

```
trim_selections(_, 0, []) :- !.

trim_selections([Sel | Selections], N, [Sel | RealSelections]) :-
    N1 is N - 1,
    !,
    trim_selections(Selections, N1, RealSelections).

% Inner loop for the min-cut algorithm.  Select a pair to be interchanged.

% Really a double do-loop.  min_cut_select is outer do -- aux is inner do

min_cut_select([], _, CostStruct, CostStruct) :- !.

min_cut_select([U0 | RestU], V, CostIn, Cost) :-
    min_cut_select_aux(V, U0, CostIn, NextCost),
    min_cut_select(RestU, V, NextCost, Cost).

min_cut_select_aux([], _, Cost, Cost) :- !.

min_cut_select_aux([V | RestV], U, cost(_,_,Cost), CostOut) :-
    computeCost(U, V, Cost1),
    Cost1 < Cost, !,
    min_cut_select_aux(RestV, U, cost(U, V, Cost1), CostOut).

min_cut_select_aux([V | RestV], U, Cost, CostOut) :-
    min_cut_select_aux(RestV, U, Cost, CostOut).

computeCost(U, V, Cost) :-
    accessField(U, nets, UNets),
    accessField(V, nets, VNets),
    ordered_set_intersection(UNets, VNets, netOrder, Nets),
    set_difference( UNets, Nets, UNets1 ),
    set_difference( VNets, Nets, VNets1 ),
    computeCostIncrement(UNets1, U, 0, CostU),
    computeCostIncrement(VNets1, V, 0, CostV),
    Cost is CostU + CostV.

% successful if name of X less than name of Y

netOrder(X, Y) :-
    accessField(X, name, NameX),
    accessField(Y, name, NameY),
    X @< Y.

computeCostIncrement([], _, Cost, Cost) :- !.

computeCostIncrement([Net | Nets], Block, CostIn, CostOut) :-
    partitionBlocks(Net, LeftBlocks, RightBlocks),
    computeIncrement(LeftBlocks, RightBlocks, Block, Inc),
    NextCost is CostIn + Inc,
    computeCostIncrement(Nets, Block, NextCost, CostOut).
```

```
partitionBlocks(Net, LeftBlocks, RightBlocks) :-
    accessField(Net, blocks, Blocks),
    splitBlocks(Blocks, LeftBlocks, RightBlocks).

splitBlocks([], [], []) :- !.

splitBlocks([Block|Blocks], [Block|LeftBlocks], RightBlocks) :-
    accessField(Block, side, Side),
    accessField(Block, selected, Selected),
    (Side = left, Selected = false; Side = right, Selected = true),
    !,
    splitBlocks(Blocks, LeftBlocks, RightBlocks).


splitBlocks([Block|Blocks], LeftBlocks, [Block|RightBlocks]) :-
    splitBlocks(Blocks, LeftBlocks, RightBlocks).

% How to compute the increment? If either side is null, block must be on the
% other side and hence moving it to this side will increase cost by 1.

computeIncrement([], _, _, 1) :- !.
computeIncrement(_, [], _, 1) :- !.

% If block is the only one on one side, moving it to the other removes this
% net from the cut. Cost decreased by 1.

computeIncrement([U], _, U, -1) :- !.
computeIncrement(_,[U], U, -1) :- !.

% Otherwise no effect on cost.

computeIncrement(_,_, _, 0) :- !.
```

12. References

[Bentley84] Bentley, J, "Programming Pearls", *CACM 27*, December, 1984.

[Bratko86] Bratko, I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1986.

[Bush87] Bush, W. and Despain, A., "High-Level Synthesis Using Prolog", *submitted to 24th Design Automation Conference*, July 1987.

[Cheng87] Cheng, G., and Despain, A., "STICKS-PAC: A Case Study in the Use of Prolog as a VLSI Design Environment", *submitted to VLSI 87*, August, 1987.

[Cohen84] Cohen, S., "Multi-Version Structures in Prolog", *Fifth Generation Computer Systems 1984*, 1984.

[Despain85a] Despain, A., and Patt, Y., "Aquarius: A High-Performance Computing System for

Symbolic/Numeric Applications", *COMPCON Spring 1985*, 1985.

[Despain85b] Despain A., "A High Preformance Prolog Co-Processor", *WESCON 1985*, 1985.

[Despain86] Despain, A., "A High-Performance Hardware Architecture for Design Automation", *Aerospace Applications of Artificial Intelligence*, 1986.

[Flanagan86] Flanagan, T., "The Consistency of Negation as Failure", *Journal of Logic Programming vol. 3; no. 2*, July 1986.

[Hill85] Hill, D., *private communication*

[Hill84] Hill, D., "...:A Case Study in the Uses of Prolog", *ICCD 84*, October, 1984.

[Knuth86] Knuth. D., "A Small Work of Literature", *CACM 29*, May, 1986.

[MacLennan83] MacLennan, B., *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart, and Winston, 1983.

[McGeer87] McGeer, P., and Despain, A. "The Topolog IC Layout Package", *submitted to VLSI 87*

[Pincus86] Pincus, J., and Despain, A., "Delay Reduction Using Simulated Annealing", *Proc. 23rd Design Automation Conference*, July, 1986.

[Tick86] Tick, E., "Memory Performance of Lisp and Prolog Programs", *Proc. Third International Conference on Logic Programming*, Springer-Verlag, 1986.

[Touati86] Touati, H., *private communication.*

[Turk86] Turk, A., "Compiler Optimizations for the WAM", *Proc. Third International Conference on Logic Programming*, Springer-Verlag, 1986.

[Warren77] Warren, D., "Implementing Prolog - Compiling Predicate Logic Programs", *Research Reports 39 & 40*, Dept. of AI, Edinburgh Univ., 1977.

[Warren83] Warren, D., "An Abstract Prolog Instruction Set", *Technical Note 309, AI Center, SRI*