PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

MULTI-VERSION STRUCTURES in PROLOG

By

Shimon Cohen

Fairchild AI lab.

4001 Miranda Ave, Palo Alto, CA 94304

ABSTRACT

In this paper we discuss the important problem of implementing MVS (Multi Version Structures) like Arrays, Hash-Tables, Sets, in logic programming Languages (i.e. PROLOG). One can define pure PROLOG predicates which behave like arrays etc. but the question is how efficient these predicates are compared to the equivalent operations in PASCAL, C or LISP. Obviously the problem in logic programming languages is that you are not allowed to change the structure of logic terms and variables' values, in logic programming you are only allowed to instantiate variables (once).
We discuss:
(1) What kind of MVS we want to have in PROLOG.
(2) The problems in implementing them.
(3) The alternative solutions.

We show how to implement arrays efficiently by introducing "Multi Version Arrays". Arrays which differ slightly from each other will be implemented using one physical array, thus the cost of updating an array while retaining the old array will be small. It is also possible (using our method) to "go back" to older versions and start modifying them (without any damage to other versions). We show how to execute parallel operations with such arrays and how to use "Multi Version Arrays" to implement sets as hashtables (not lists). Sets are important and diverse data structures with many special cases, in order to take advantage of this phenomena we propose to add some specifications (while "creating" the set) which will enable the system (compiler) to choose the most efficient internal representation (The internal representation will be transparent to the user).

* Part of this work was done while the author was a visitor at UC Berkeley.

1. Introduction

The goal of this research is to investigate and improve the way logic programming languages handle Multi Versions Structures (MVS). In first order logic there are logical terms whose structure cannot be changed and variables which can be instantiated only once. For reasons of efficiency, we would like to introduce arrays sets etc. The problem with these MVS is that when you use them to create new versions you need to copy the entire structure. (i.e. When you change an element of an array you need to copy the entire array). In the sequel: Given an array A, if you change one of it's entries and create a new array B then we call A "the old version" and B "the new version". In this paper we discuss the difficulties of, and alternative solutions to, this problem.
There are several things you want to take care of when you deal with mutable arrays:
(1) The access time (of one element) in the newest array.
(2) The access time (of one element) in older versions.
(3) The update time.
(4) The Ability to modify old versions of the array while retaining new ones.
(5) Execute parallel operations on some (or all) of the elements of the array.
(6) Efficient representation of SPARSE arrays.
(7) Dealing with Garbage Collection.
NOTE: Some of these requirements are contradictory.

1.1. Related Work

The "tree method" <Okeefe 84> (see the listing in the appendix A) is based on the breadth first search method of a binary tree. Each node, in the tree, has an element of the array and two pointers to other elements. The root of the tree is index number 1, his left child is No. 2 and his right child is No. 3. The left/right children of No. 2 are 4/5 and for No. 3 are 5/6 etc. Given an index (N) number here is the algorithm that will get you to the desired node which holds element a[N]:
(NOTE: Look at N as a bit vector (left most bit is the most significant bit))

(a) Search (from the left) for the first 1 bit.
(b) Skip this bit and look at the bits to it's right.
(c) Use the rest of the bits (down to the less significant bit) as instructions
    zero - go to the left.
    one - go to the right.

For example: 6 is 00000...00110
(a) The first 1 bit is the third from the right.
(b) Skip it and you look at the second bit form the right.
(c) Go right (second bit is 1) and then left
    (rightmost bit is zero).
(d) Congratulation !! you made it.

When you implement a sparse array on top of this tree, you need only to maintain the nodes on the path to existing indexes. Pereira noted <Pereira 84> about the tree method: "The real beauty of the log (tree) methods is the space efficient implementation of SPARSE arrays: big unused holes need not be there at all".

My comments are: (1) I never argue with people about beauty. (2) I agree that the tree method is space-efficient for SPARSE arrays. (3) In many cases arrays are dense (for example: when they are used to implement hash-tables) and then you pay two pointers per element as overhead.
KAHN < KAHN 84 > proposed (in parallel to this work) to use one physical array to hold the newest version of the array. Other old versions of the array are maintained by lists of "value blocks". Each value block contains index-value pair which enable the system to restore old values. An array is represented as a list of changes which ends with a pointer to the physical array. The physical array contains the newest version. When you update the array you copy the ENTIRE list of changes (not very efficient for old arrays) and add the last index value (before the update) to the list of the old arrays. His method works fine O(1) when you want to access or modify the newest version. BUT it works very bad O(number of changes) if you want to access/update previous versions. It is not clear whether his method is amenable to parallel operations.

We propose a different method which allows the user to use the same physical array for many "slightly" different arrays. The cost for accessing an element in the most new array (in our method) is O(1) meaning: it does not depend on the size of the array or the number of changes or versions which use the same physical array. The cost of update (in our method) is ALLWAYS O(1) even if you change very old versions. The cost for accessing an element in older versions of the same array is on the average O(number of changes to the element) compare it with O(number of changes to the array) in KAHN method.

In our method you can also take an old version of the array and modify it (creating a tree of versions) while you still use the same physical array.
The average here is: O(log(number of changes) + number of changes to the particular element)
The worst case will be: O(half the number of changes + changes to the particular element)

One advantage of KAHN method is the fact that his array package was implemented on top of the the LISP Machine as part of the LM PROLOG and the way it was implemented needed no changes to the Garbage Collector. The 'tree' method can be written in pure PROLOG (again see the appendix) so there are no special problems for the GC. I suspect that the method we propose need some arrangements with the GC

Summary of different proposals

The tree method -
* It is not so good for access/update of elements in any version O(log(number of elements)).
* It is good for SPARSE arrays (space) but not for dense array (three times overhead).
* Old versions are not destroyed or Garbage Collected unless you discard them.
* Garbage Collection is easy.
* Can be implemented in pure PROLOG.
* You can easily increase the size of the array.

KAHN method -
* It is good for update/access of elements in the most new version of the array.
* Old versions are painful O(number of updates).
* It is possible to update old versions and have different versions.
* It is not efficient (space) for SPARSE arrays.
* It is easy to GC them.
* It is hard to extend the size of the array, you need to copy the entire array.

Multi Version Method (our method) -
* It is good as KAHN method and better then the tree method for update/access elements in most new array.
* It is better in accessing elements in old versions. O(number of changes to the element accessed)
* It is the best in updating old versions (allways constant time).
* It is as bad/good as KAHN method for SPARSE array (worse then the tree method).
* You need to "teach" the Garbage-Collector about this new data-structure.

1.2. Where is the problem ?

To illustrate the problem suppose we have the following primitive operation in PROLOG:
array_update(A, I, V, NewA).

The meaning of this predicate is: "NewA is the array A whose I-th element is changed to V".
You don't want to copy A every time you create NewA, you also want to be able to execute:
array_update(A, I1, V1, A1), array_update(A, I2, V2, A2).

It means that arrays A1 and A2 are both derived from array A, and now we have three (A, A1, A2) arrays each one of them is slightly different from the others.

Once you have "Multi Version Arrays" in PROLOG you can implement sets quite efficiently as hashtables. We are interested in implementing the following clause:
oneof(S, E, Sr).

Read it:
"E is an element of the set S and Sr is the remaining set".
It can be used for the following problem: " select three different people from a group "
then you would like to execute:
oneof(S, Person1, S2), oneof(S2, Person2, S3), oneof(S3, Person3, S4).

Note that the set S is unaffected by the above execution, at the same time you can use S4 as the set of people without person1-3.

Right now (in PROLOG) sets are represented as lists and you have to use list operations to achieve the same effect:

```
oneof( [E | Sr], E, Sr).
oneof( [A | Sx], E, [A | Sr]) :- oneof(Sx, E, Sr).
```

However, if E is a term which is partially instantiated and the set is big it will be much faster to use another structure (maybe hashtable) and another technique (hashing) to locate E. The exact efficient internal representation of a set depends on it's size and the type of elements etc. Another Example: Each PROLOG system has a database of facts and rules, this database is a set, can you imagine using list representation for this set ??. It is unthinkable to access elements in this set using list operations.

We claim that sometimes you want a general-type set, BUT sometimes you have additional information which might help the system/compiler to decide on the appropriate efficient representation. We therefor propose a way to specify additional knowledge you might have about sets. An attempt in this direction was the SETL 'base' declaration <SETL 75>

2. Extensions

In this section we propose the following extensions to PROLOG:

2.1. Sets

Instead of using the *setof* or *bagof* functions which do not return "sets" or "bags" (but a list of all elements in the "set" / "bag") we propose to introduce a rather familiar data-type namely a **set**. A real 'set' will be implemented in the most efficient way (most of the time as an hash-table, BUT maybe as a list or bit array) depending on it's characteristics. The way sets will be used is as follows:
(1) **set_of_all(E, P, S).**

Which is the same as the familiar *setof* function except that S is a "real" set.
(2) **oneof(E,S).**

Read it as follows:" E is a member of the set S".
The problem in (2) is that we can't refer to the rest of the set S, to allow it we introduce:
(3) **oneof(E,S,Sr).**

There are some interesting cases:
CASE 1 - Only S is instantiated.
Then: E is an element of the set S where Sr is the new set resulting from the deletion of E from the set S.
CASE 2 - S and E are instantiated.
Then: if E is in S then as the above otherwise fail.
CASE 3 - Sr and E are instantiated.
Then: S is the result of adding E to the set Sr.
CASE 4 - Only E is instantiated.
Then: Sr is the empty set and S is the single (E) element set.

If S is a list (as in ordinary PROLOG) which was generated, for example, by *setof* then one can easily access the first element (and the rest of the elements) by using list constructor: [E | Sr]. Sets as lists are very simple but can become very inefficient when dealing with big sets (accessing random element) or preforming big set operations like: intersection or union.
Lets su see now how we create (declare) new empty sets:

(4) **set(S,Specs).**

Read this as follows: "S is the set with the Specs specifications" where Specs is a list with the following possible items (non is required):
size(N) -
Approximated size of the set (the actual size can be bigger)
The compiler may decide on the size of the hash-table.
integer(N1,N2) -
The set consists of integer numbers in the range of N1..N2.
The compiler may decide to implement the set as a bit array.
Example: **integer(0,100).**
list(L) -
L is a list of atoms that serve as the base set for the set S.
Example: **list([sunday, monday, tuesday, wednesday, thursday, friday, saturday])**
base(BaseS) -
BaseS is an existing set which is the base set of the set S.
Example: **base(Sx)** where Sx was defined earlier as a set.
map(Element, Key) -
Elements of the set have two parts: Key and Data.
For example: **map(a(i,v), i).**
In the example: elements are 'a(i,v)' and the key is 'i'.
If you couple it with the other (see above) specs. then you can define an array as a special case:
set(S, [map(a(i,v), i), integer(i,1,100)]).
will be as in pascal: ' array [1..100] of v '.

2.2. Arrays

Arrays are going to be used as follows:
(1) **is_array(A, Size)**
(a) If *A* & *Size* are instantiated then it is a predicate.
(b) If only *A* is instantiated then *Size* gets the size of *A*.
(c) If only *Size* is given then *A* is a new array of size *Size*.
(2) **array_element(A, Index, Value).**
— *A* and *Index* must be instantiated. —
(a) If *Value* is given then it is a predicate.
(b) If *Value* is a variable then it gets the value of the index-th element.
(3) **array_update(A, Index, Value, NewA).**
— Except for *NewA* all the other variables must have values.
NewA is an array which is like array *A* except that element *Index* is replaced.

is_array can be called with an extra parameter called: **specs**
This can take any of the following values:
sparse - sparse array.
new - new version of array is most likely to be used.
onlynew - Only newest version will be used for update and access.
onlylast - Only last version will be UPDATED.
all - all versions of the array are going to be used (default).

These **specs** (as in the set case) can help the system to determine the right internal representation.

3. IMPLEMENTATION

In this chapter we discuss the implementation of "Multi Version Arrays" (MVA) and the way we can operate in parallel on them.

3.1. MVA

The idea is to use one "Physical Array" to represent several slightly different arrays. Since these arrays have a lot in common, we say that they are different "versions" of the same array. The "Physical Array" is implemented using one physical array with the following information:
(1) MVAC - (Multi Version Array Counter) A counter which counts the number of updates, initially it is zero and it is incremented in every update.
(2) EL(i) - (Element List) Each element in the array is a list of pairs, every time we update entry *i* we push the pair (Version . Value) onto the front of EL(i). Another words: for each element we keep a list of updates and the time (version) they were made.
To represent different versions (with the same physical array) we use MVAPL - (Multi Version Array Period List).
MVAPL consists of: (1) a pointer to the physical array (2) A "period List" which is a list of pairs of numbers. Each pair represents a period of updates which are valid for this version of the array.

EXAMPLE

Suppose we have an array A with three elements initialized to NIL:

Physical Array (call it P):

```
MVAC: 0
[1] : nil
[2] : nil
[3] : nil
```

MVAPL of array A: (<ptr to P> ((0 . 0)))

Now we create a new array B by updating A[1] to v1:

Physical Array (call it P):

```
MVAC: 1
[1] : (( 1 . v1))
[2] : nil
[3] : nil
```

MVAPL of array B: (<ptr to P> ((1 . 0)))
(NOTE: MVAPL of A remains the same):

Now we create a new array C by updating B[2] to v2:

Physical Array (call it P):

```
MVAC: 2
[1] : (( 1 . v1))
[2] : (( 2 . v2))
[3] : nil
```

MVAPL of array C: (<ptr to P> ((2 . 0)))

(NOTE: MVAPL of A,B remain the same):

Now we create a new array D by updating C[1] to v3:

Physical Array (call it P):

```
MVAC: 3
[1] : (( 3 . v3)( 1 . v1))
[2] : (( 2 . v2))
[3] : nil
```

MVAPL of array D: (<ptr to P> ((3 . 0)))
(NOTE: MVAPL of A,B,C remain the same):

Now array D is the newest version of the array but using MVAPL of array B (for example) we can still access the right value of B[1] which is v1 (not v3) because the MVAPL of B tells us that only updates made in the period 0..1 are valid for version B. NOTE that as long as we keep taking the newest version and update it to create a new version THEN the size of the new MVAPL remains the same.

Now we create a new array E by updating C[1] to v4: (NOTE that we take an old version C and not the most new version D)

Physical Array:

```
MVAC: 4
[1] : ((4 . v4)( 3 . v3)( 1 . v1))
[2] : (( 2 . v2))
[3] : nil
```

MVAPL of array E: (<ptr to P> ((4 . 4)(2 . 0)))

The "Period List" of E consists two periods 0..2,4..4 because array E is derived from array C and changes in period 3 are not valid for it.

3.2. Update algorithm

Here is the algorithm for a single update in MVA. The full listing is given in the appendix B, the algorithm is written in LISP because it uses side effect operations (pushing a pair onto the EL(i) lists).

array_update(OldMVAPL, Index, NewValue, NewM-VAPL).

```
(1) Add 1 to MVAC. ; must be distructive
(2) Push the pair (MVAC . NewValue) onto EL(Index).
(3) Create a new MVAPL (new version of the array)
    Given that OldMVAPL is:
      [ [MVAC,Array] , [V,Vstart] | RestMVAPL ].

      where:
        ARRAY - physical array.
        V - top version number.
        RestMVAPL - rest of the Old Array Skip List.

 (3.a) If (MVAC = V+1) then the new NewMVAPL is:
   [[MVAC, Array], [MVAC, Vstart] | RestMVAPL ]
 (3.b) If (MVAC > v+1) then we update old version of the
array then MVAPL is:
   [[MVAC, Array], [MVAC, MVAC], [V, Vstart] |
RestMVAPL].
```

NOTE:
(a) Update always takes a constant time.
(b) The size of the MVAPL remains the same (3.a)
Unless you go back and update an old version (3.b) of the array.
(c) There is always a pointer to the "physical array" in the beginning of the MVAPL.

Here is the algorithm in PROLOG given (for god sake NO ...) that we have the "dirty" operation:
change_entry(Array, Index, Value)
which distructively changes the array.

```
array_update( [ [MVAC,Array] [V,Vstart]| RestMVAPL],
                        Index, Value, NewA) :-
   MVAC1 is MVAC + 1,
   change_var( MVAC, MVAC1 ), ; side effect
   get_entry( Array, Index, El), ; get ELIST
   change_entry( Array, Index, [ [ MVAC1, Value ] | El]),
   make_MVAPL( Array, V, Vstart, MVAC, MVAC1,
                     RestMVAPL, NewA).

make_MVAPL( Array, V, Vstart, V, MVAC1, RestMVAPL,
     [ [MVAC,Array], [MVAC1,Vstart] | RestMVAPL]) :- !.
   ; MVAC = V then just replace V with MVAC1
make_MVAPL( Array, V, Vstart, MVAC, MVAC1, RestMVAL.
     [ [MVAC,Array],[MVAC1,MVAC1],[V,Vstart] | RestMVAPL]).
   ; add a new pair to the MVAPL list.
```

3.3. Access Algorithm

Here we describe the algorithm which is used to access an element in the array. Before we go into the details of this algorithm we will give some examples:

Example 1:
MVAPL is: [[20, #array], [20,0]] ;
Index: 3
EL(3): [[18, v1], [12, v2], [5, v3]]
THEN: v1 is the result.

Example 2:
MVAPL is: [[20, #array], [20, 19], [15,0]]
It means: Only versions in the range [0..15,19..20] are good for this MVAPL.
Then: v2 is the result.

Example 3:
MVAPL is: [[20, #array], [20, 19], [15, 13], [7, 0]]
It means: Only versions in the range [0..7,13..15,19..20] are good.
Then: v3 is the result.

array_element([Array | RestMVAPL], Index, Value).

```
; PLIST and ELIST are local vars
                    The Access algorithm
 (1) Set PLIST <-- RestMVAPL, ELIST <-- EL(Index).
 (2) If ELIST is empty return nil (default value)
 (3) If the first version number in PLIST is less then
       the version number of the first value in ELIST
       then: skip that element from ELIST and continue in (2).
 (4) Now we know that the end-of-period version number is
 bigger so we want
       to check if the start version number is less or equal to
 the element version.
       If it is so then this is the element.
 (5) Otherwise: we skip the top pair (of version numbers)
 from PLIST and continue in (2).
```

Here we all have a chance to figure out whether it will be more clear to grasp the algorithm when it is written in PROLOG:

```
array_element( [ Array | RestMVAPL ], Index, Value) :-
  get_entry(Array, Index, ELIST),
   ; primitive access to array EL(Index).
  array_find( RestMVAPL, ELIST, Value).

array_find(_, [], []). ; default is null list
array_find( [ [V, Vstart] | Rasl], [[ Ev, _ ]
   | Rellst], Value) :- V < Ev , !,
   array_find( [ [V, Vstart] | Rasl ], Rellst, Value).
   ; drop first pair from ELIST
array_find( [ [V , Vstart] | RestAPL], [[ Ev, Value ]
   | Rellst], Value) :-
    Vstart <= Ev , !. ; Value is returned (see unification)
array_find( [ [V, Vstart ] , [[ Ev, Value ] | _] , Value) :- !.
   ; special case when the last version in PLIST is alone...
array_find( [ _ | Rplist], Elist, Value) :-
   array_find( Rplist, EList, Value).
   ; Drop a pair of version numbers from PLIST and continue.
```

3.4. Backtracking

In Backtracking we want to undo the effect of the last update; since update is a simple operation the opposite (backtracking) is also simple, actually it is simpler. To backtrack do the following steps:
1. Decrement the counter MVAC (of the physical array).
2. Discard the top pair in EL(Index) where Index is the last updated entry.

To do that we need to keep on the backtracking list a record with: (1) the Index (2) a pointer to the physical array. Once we backtrack into this record, we simply do the above O(1) operation.

3.5. Doing it in parallel

Suppose we want to apply a function/predicate P in parallel to an array and change some (or all) of it's entries. The predicate P has two parameters **P(input,output)**, we apply it to each entry (which is the input) and replace it with the output (if P succeeds), If P fails then the old value remains, Example: Suppose we have a big matrix and we use a simple relaxation algorithm which iteratively changes the element value until a termination condition is met. This predicate is applied in parallel to all the elements in the matrix.
Using the MVA technique we can do the following:
1. Add one to the MVAC.
2. Use this number in the parallel application as the version number of the value. Recall that each time we update a value we add (infront of the EL(i)) a pair [Version , Value].
The algorithm in Concurrent PROLOG <Shapiro 83> (for example) will be: (NOTE that we are going to have side effect operations again)

```
  parallel_map( P, MVAarray) :- | ; commit (why not ???)
    MVAarray = [ [MVAC, Array] | Plist],
    MVAC1 is MVAC + 1,
    change_var(MVAC, MVAC1),
    mvasize( MVAarray, Size),
    map_in_para(P, 1, Size, Array, MVAC1),
    make_MVAPL( Array, MVAC, MVAC1, Plist, NewA).

 make_MVAPL( Array, MVAC, MVAC1, [[MVAC,Vstart]Rest],
   [ [MVAC1,Array], [MVAC1,Vstart] | Rest]) :- |.
 make_MVAPL( Array, MVAC, MVAC1, [[Vold,Vstart]Rest],
   [ [MVAC1,Array], [MVAC1,MVAC1],[Vold,Vstart] | Rest]) :- |.

 map_in_para(P, From, To, Array, MVAC1) :-
  From !== To, |,
  map_in_para(P, From , To2, Array, MVAC1),
  map_in_para(P, To21, To, Array, MVAC1).
; While 'doit' does it in parallel

 map_in_para(P, Index, Index, Array, MVAC1) :- |.
    array_ELi(Array, Index, ELi),
    doit(P, Current_value, Index, Array, MVAC1,Eli),

 doit(P, Current_value, Index, Array, MVAC1, El) :-
    P(Current_value,New_value),|, ; success of P means change
    change_entry( Array, index, [ [ MVAC1, Newvalue ] | El]).
 doit(_,_,_,_,_,_) :- otherwise,|.  ; If no need to change don't fail,
return safely.
```

3.6. Trade-offs Optimizations

As it often happens in real life, we can use simple trade-offs and some practical considerations to achieve efficient behavior of the program. In this case we want to insure that worst case access time per element will ALLWAYS be O(number of changes to this element).
We observe that MVAPL is a list of pairs of versions numbers, each pair represents a period of time in which updates are significant. Actually MVAPL can be represented as a set (bit vector of size MVAC) where the K-th bit represent version number K. If the bit is 'on' then values updated with this version number are significant for this MVAPL. Maintaining this bit vector imposes changes on the update algorithm:
(1) Increment the MVAC of the physical array.
(2) Copy the bit vector from the old MVAPL
 (which is also a bit vector).
(3) Increase the size of the bit vector by one
 (bit) and set it to 'on'.
(4) Add the pair (MVAC - Value) to the element list.

Obviously in step no. 2 we are doing O(number of changes) operations which is worse then the O(1) (that we previously claimed for the update operation), BUT we gain alot in terms of access (for the worst case) which becomes O(number of changes to the accessed element). ALSO practically every 100 updates need roughly 3 long words (assuming 32 bits per word). In KAHN method (in the worse case) you will need to copy 100 values ... If we update in parallel then steps (1)-(3) are executed once for the entire operation and then each process updates it's own element.
The access algorithm is simplified: If you access the K-th element then you simply iterate down EL(K) and use the version number of each version-value pair to check the bit in the bit vector MVAPL. If the bit is 'on' then this is the required value. NOTE that access time for the most new array is still O(1).

Finally, you can use a mixture of these two representations (1) list of periods (pair of version numbers) (2) bit vector. Use a list of bit vectors each of them 32 bits long (one long word). Each element in the list will be eight bytes long divided as follows:
(1) 4 bytes for the bit vector.
(2) 1 byte for number of bit vector.
(3) 3 bytes pointer to the next bit vector.

That will be good enough for 256 * 32 updates (total of 8196 updates), after which we simply recommend to (really) copy the entire array and start from the beginning (see appendix C).

Other Special Cases

If we can guarantee (using a compiler, for example) that the array is going to be used as in PASCAL or C meaning: only the last version is accessed or modified, then we can simply use arrays with side-effects (Big deal..). If we can guarantee that only the most new version is going to be modified but we allow access to old versions then we can do the following:
A pointer of a given version of the array will be a pair: (1) version (2) pointer to the physical array.
Each entry of the physical array will have two fields:
(a) NEWvalue - the newest value.
(b) EL(i) - the list of pair (as before).

UPDATE -
(1) Increment MVAC by 1.
(2) Push the pair: MVAC, NEWvalue(i) to EL(i).
(3) Insert updates value into NEWvalue.
(4) Return a pointer to the array which is a pair: MVAC, address.

ACCESS - (newest version)
As fast as PASCAL or C, use the array address and access the value in NEWvalue field.
ACCESS - (old version)
Use the version number in the array to look for the value in EL(i).

4. Summary

In this paper we addressed ourselves to the problem of implementing MVS. We propose how to define the appropriate predicates in the language and then we discussed the problems and solutions.

The range of array applications can be categorized by:
* Dense vs. Sparse arrays.
* Size of arrays.
* Update of last version vs. update of all versions.
 (most of the time/ only)
* Access of last version vs. access of all versions
 (most of the time).

The alternatives for internal representation are:
* A copy per version - Need to copy the entire array every update.
* Tree method - good for Sparse array.
* Ken Method - good for array where the last version is used most of the times.
* MVS method - good when all versions are likely to be accessed.
* One Physical array - Only if you are sure that you will access the newest version.

5. References

<Clark 82>
Clark K. L. and McCabe F. G.
"PROLOG: a language for implementing expert systems"
Machine Intelligence 10, 1982.

<Carlsson 83>
Carlsson M. and Kahn K.
"LM-Prolog User Manual",
UPMAIL Technical Report No. 24,
Uppsala University, Sweden, Nov. 1983

<Clocksin 81>
Clocksin W. and Mellish C.
"Programming in PROLOG"
Springer-Verlag, Berlin, Hiedleberg, NewYork 1981.

<Griswold 83>
Griswold R. E.and Griswold M. T.
"The ICON Programming Language"
BOOK, Prentice Hall, 1983.

<Hill 74>
Hill R.
"LUSH resolution and its completeness"
DCL Memo NO. 78, Dept. of Artificial Intelligence,
Univ. of Edinburg, 1974.

<Kahn 84>
Kahn K.,
"Incorporating Mutable Arrays into Logic Programming"
Private Communication,
The Intl. Conf. on Logic Programming in Sweden 1984.

<Kennedy 75>
Kennedy K. & Schwatrz J.
"An Introduction to the Set Theoretic Language SETL"
Computation and Mathematics with Applications, 1 (1975)
pp. 97-119.

<Kowalski 74>
Kowalski R. A.
"Predicate Logic as Programming Language"
Proceeding IFIP Congress 1974.

<McDermott 80>
McDermott D.
"The PROLOG phenomenon"
SIGART Newsletter 72, pp:16-20, 1980.

<Okeefe 84>
R.A. O'Keefe
"Updateable Binary Trees"
In Stanford On-line PROLOG library (SU-score)

<Periera 84>
Periera F.
A comment about mutable arrays which appeared
in the PROLOG electronic digest

<Robinson 82>
Robinson J. A. and Sibert E. E.
"LOGLISP: An alternative to PROLOG"
Machine Intelligence 10, 1982.

<Sato 83>
Sato M. and Sakurai T.
"QUTE: A PROLOG/ LISP type language for
LOGIC programming"
Proc. of 8th intl. Joint Conf. on AI, Volume 1,
August 83, Karlsruhe, West Germany.

<Shapiro 83>
Shapiro E.
"A subset of Concurrent PROLOG and it's Interpreter"
ICOT Technical Report, TR-003, ICOT 1983.

<Warren 77>
Warren D. H. D., Periera L. M. and Periera F.
"PROLOG- The language and its implementation compared
with LISP"
Proc. of Symp. on AI and Programming Languages, 1977,
SIGPLAN Notices, 12, No. 8, and SIGART 64, 109-115.

<Warren 80>
Warren D. H. D.
"Logic Programming and Compiler Writing"
Software Practice and Experience 10, pp: 97-125, 1980.

<Warren 82>
Warren D. H. D.
"High order extension to PROLOG: are they needed ?"
Machine Intelligence 10, 1982.

Appendix A

```
/*

Here is a copy of O'Keefe file for updateable arrays (the
tree method). in the end we added pp_tree & top which help
you to test and understand this method.
 File   : TREES.PL
 Author : R.A.O'Keefe
 Purpose: Updatable binary trees.

  We have:
    list_to_tree : O(N)
    tree_to_list : O(N)
    get_label    : O(lg N)  as: array_element
    put_label    : O(lg N)  as: array_update

Where N is the number of elements in the tree. The way
get_label and put_label work is worth noting: they build up a
pattern which is matched against the whole tree when the
position number finally reaches 1. In effect they start out
from the desired node and build up a path to the root. They
still cost O(lg N) time rather than O(N) because the patterns
contain O(lg N) distinct variables, with no duplications.
put_label simultaneously builds up a pattern to match the old
tree and a pattern to match the new tree.
 get_label(Index, Tree, Label)

treats the tree as an array of N elements and returns the
Index-th. If Index < 1 or > N it simply fails, there is no
such element.

*/

get_label(N, Tree, Label) :-
    find_node(N, Tree, t(Label,_,_)).

    find_node(1, Tree, Tree) :- !.
    find_node(N, Tree, Node) :-
       N > 1,
       0 is N mod 2,
       M is N / 2, !,
       find_node(M, Tree, t(_,Node,_)).
    find_node(N, Tree, Node) :-
       N > 2,
       1 is N mod 2,
       M is N // 2, !,
       find_node(M, Tree, t(_,_,Node)).

/*

list_to_tree(List, Tree)
takes a given List of N elements and constructs a binary
Tree where get_label(K, Tree, Lab) <=> Lab is the Kth
element of List.

*/

list_to_tree(List, Tree) :-
 list_to_tree(List, [Tree|Tail], Tail).

 list_to_tree([Head|Tail], [t(Head,Left,Right)|Qhead],
   [Left,Right|Qtail]) :- list_to_tree(Tail, Qhead, Qtail).

 list_to_tree([], Qhead, []) :- list_to_tree(Qhead).
 list_to_tree([_|Qhead]) :- list_to_tree(Qhead).
 list_to_tree([]).
```

/*

put_label(Index, OldTree, Label, NewTree)

constructs a new tree the same shape as the old which moreover has the same elements except that the Index-th one is Label. Note that O(lg N) new space is needed.

*/

```
put_label(N, Old, Label, New) :-
 find_node(N, Old, t(_,Left,Right), New,
                    t(Label,Left,Right)).

 find_node(1, Old, Old, New, New) :- !.
 find_node(N, Old, OldSub, New, NewSub) :-
  N > 1,
  0 is N mod 2,
  M is N  /  2, !,
 find_node(M, Old, t(Label,OldSub,Right), New,
                    t(Label,NewSub,Right)).
 find_node(N, Old, OldSub, New, NewSub) :-
  N > 2,
  1 is N mod 2,
  M is N  //  2, !,
  find_node(M, Old, t(Label,Left,OldSub), New,
                    t(Label,Left,NewSub)).
```

/*

tree_to_list(Tree, List)

is the converse operation to list_to_tree. Any mapping or checking operation can be done by converting the tree to a list, mapping or checking the list, and converting the result, if any, back to a tree. It is also easier for a human to read a list than a tree, as the order in the tree goes all over the place.

*/

```
tree_to_list(Tree, List) :- tree_to_list([Tree|Tail], Tail, List).

 tree_to_list([], [], []) :- !.
 tree_to_list([A|_], _, []) :- var(A),!.
 tree_to_list([t(Head,Left,Right)|Qhead],
   [Left,Right|Qtail], [Head|Tail]) :-
       tree_to_list(Qhead, Qtail, Tail).
```

/*

pretty print the tree (pp_tree was written by myself).
Use it to print the tree created by 'list_to_tree' etc.

*/

```
pp_tree( Tree ) :- pp_tree( Tree, [], t).
pp_tree( Tree, _,_ ) :- var(Tree),!. % empty leaf
pp_tree( t(Label,L,R), Ll, LRT) :-
 lrt(l,LRT,YesNo1),
 pp_tree(R, [YesNo1|Ll],r),
 ttline(Ll),write(Label),nl,
 lrt(r,LRT,YesNo2),
 pp_tree(L, [YesNo2|Ll],l).

 ttline([]) :- !.
 ttline([y|L]) :- ttline(L),write('| ').
 ttline([n|L]) :- ttline(L),write('  ').

 lrt(LRT,LRT,y) :-!.
   % returns 'y' if first two params are the same
 lrt(_,_,n). % otherwise 'n'

% top - simple top loop to test the above defs.

top :- top(Tree).
top(Tree) :-
 repeat,
 write('Enter: a(I,V). (NOTE q. to exit) '),nl,
 read(Input),
 top(Tree,Input).
top(Tree,q) :- !. % cefini
top(Tree,a(I,V)) :-
 put_label(I,Tree, V, NewTree),
 write(' Tree after update ') ,nl,
 pp_tree(NewTree),
 top(NewTree).
```

Appendix B

/*

This is the LISP version of the MVA package, the main reason for writting it in LISP is the need for side effects. NOTE: this file was tested on FRANZ lisp.

Multi version arrays for LISP systems (This package is for FRANZ LISP)

Written By: Shimon Cohen
Time: Feb 4, 1984
Place : UC Berkeley

NOTE: To help implement this package in another LISP system we list the functions which seem to be unique to FRANZ LISP. (In FRANZ LISP the array we are using is called 'vector'): The rest is written in "standard" LISP which will (hopefully) be portable without modification.

```
(new-vector 'size)
    RETRUNS: Internal LISP array of size 'size'
(vset 'mvarray 'index 'value)
    RETURNS: the value
(vref 'mvarray 'index)
     RETURNS: The element 'index
```

The above functions are used by this package to implement the following MVA (Multi Version Array) package:

```
(mvaarray 'size)
   RETURNS: A new array of size 'size'.
(mvap 'array)
   RETURNS: true if 'array' is mva array.
(mvaset 'oldarray 'index 'value)
   RETRUNS a new array (the old one retains it's values)
(mvaref 'array 'index)
   RETURNS: the value of 'index' element.
(mvasize 'array)
   RETURNS: The mva array size.
(mvachanges 'array)
   RETURNS: The number of modifications made to the
```

```
array.
 (mvacopy 'array)
    RETURNS: a copy of the array (without old history)

*/

 (def mvaarray (lambda (size)
  (cons (cons 0 (new-vector size))
       (list '( 0 . 0)))))

 (def mvap (lambda (array)
   (and (listp array)
        (listp (car array))
        (listp (cdr array))
        (numberp (caar array))
        (vectorp (cdar array)))))

 (def mvasize (lambda (a) (vsize (cdar a))))
 (def mvachanges (lambda (a) (caar a)))

/*

A pointer to a mvaarray has the following structure: The
'car' points to a dotted pair whose 'car' is the 'clock' and the
'cdr' is the actual array.  The 'cdr' is a list of pair of numbers
(time periods).

*/

 (def mvaset (lambda (oldarray index value)
   (prog (clock   ; internal array "clock"
     clock1 ; plus 1
     a      ; "internal array" (a real one)
     l      ; history list
     cl     ; dotted pair f the above
      )
     (setq cl (car oldarray))
     (setq clock (car cl))
     (setq clock1 (add1 clock))
     (setq a (cdr cl))
     (setq l (cdr oldarray))
     (vset a index (cons (cons clock1 value)
                        (vref a index)))
     (rplaca cl clock1) ; update "internal clock"
     (return (cons cl
             (cond ((eq clock (caar l))
                    (cons (cons clock1 (cdar l))
                         (cdr l)))
                   (t (cons (cons clock1 clock1) l)))
        ))))); end of MVASET function

 (def mvaref (lambda (array index)
   (mvaref-find
        (cdr array) ; the pointer history
        (vref (cdar array) index) ; The element history
        )))

(def mvaref-find (lambda ( plist elist )
 (prog  nil
  loop
   (cond
     ((null elist) (return nil))
     ((< (caar  plist) (caar elist))
```

```
       (setq elist (cdr elist))
       (go loop))
      (( > = (caar elist) (cdar plist)); the element "time"
       (return (cdar elist)))
      (t (setq plist (cdr plist))
         (go loop)))
      ))) ; end of MVAREF-FIND

/*

 MVACOPY
 Fast Copy of 'mva array'

*/

(def mvacopy (lambda ( oldarray )
   (prog (newarray i size a v)
         (setq size (mvasize oldarray))
         (setq newarray (mvaarray size))
         (setq a (cdar newarray))
         (setq i 0)
     loop
         (cond ((eq i size) (return newarray)))
         (setq v (mvaref oldarray i))
         (cond ((null v) nil) ; default is nil anyway ...
               (t (vset a i (list (cons 0 v)))))
         (setq i (add1 i))
         (go loop))
     )) ; end of MVACOPY


(def mvaprint (lambda ( array flag )
   (prog (i size a v)
         (setq size (mvasize array))
         (setq a (cdar array))
         (setq i 0)
     loop
         (cond ((eq i size) (return array)))
         (print i)
         (patom ": ") ; prints without quotes marks
         (cond (flag (print (vref a i)))
               (t (print (mvaref array i))))
         (terpri)
         (setq i (add1 i))
         (go loop))
     )) ; end of MVAPRINT
```

Appendix C

```
/*

Load this file after you load the file in Appendix B, then use
the same old functions: mvaset & mvaref the same way.
These Procedures were optimized to squeeze adjacent bit
vectors of all 1.
Such adjacent bit vectors are replaced by a single number (1).
A pointer to a mvaarray has the following structure: The
'car' points to a dotted pair whose 'car' is the 'clock' and the
'cdr' is the actual array.
The 'cdr' is a list of pair of numbers where:
 cdr - a 32 bit vector
 car - number of bit vector

*/

(def mvaset (lambda (oldarray index value)
 (prog (clock  ; internal array "clock"
    clock1 ; plus 1
    a      ; "internal array" (a real one)
    l      ; history list
    cl     ; dotted pair of the above
    cmod   ; clock mode 32
    cnum   ; clock div 32
     )
  (setq cl (car oldarray))
  (setq clock (car cl))
  (setq clock1 (add1 clock))
  (setq a (cdr cl))
  (setq l (cdr oldarray))
  (setq cmod (mod clock 32))
  (setq cnum ( / clock 32)) ; shift right 5 bits
  (vset a index (cons (cons clock value) (vref a index)))
  (rplaca cl clock1) ; update "internal clock"
  (return
    (cons cl
      (if (eq cnum (caar l))
          ; if still space in this bit vector
         (cons (cons cnum
                (boole 7 (cdar l) (lsh 1 cmod))) ; OR in
             (cdr l))
        (cons (cons cnum (lsh 1 cmod))
                 ; new bit vector 32 bit
          (if (and (eq -1 (cdar l))
                   ; -1 means; all bits are 1
                  (eq (caar l) (sub1 cnum)))
                     ; squeeze all 1-s bit vectors
            (if (cdr l)
              (cons 1 (if (eq 1 (cadr l))
                          (cddr l) (cdr l)))
              (if (zerop (caar l)) (cons 1 (cdr l)) l))
                 l))))) ; end of return
            ))) ; end of MVASET function

(def mvaref (lambda (array index)
  (mvaref-find
     (cdr array) ; the pointer history
     (vref (cdar array) index) ; The element history
      )))

(def mvaref-find (lambda ( plist elist )
 (prog  (flag mv pv)
   (setq flag nil)
  NEXTplist
   (if (eq (car plist) 1)
     (progn (setq flag t) (setq plist (cdr plist)))
     (setq flag nil))
  NEXTelist
   (if (null elist) (return nil))
   (setq mv (mod (caar elist) 32))
   (setq pv ( /  (caar elist) 32))
   (if flag (if (or (null plist) (>  pv (caar plist)))
                 (return (cdar elist)))
     (if (>  pv (caar plist))
        (progn (setq elist (cdr elist))
              (go NEXTelist))))
   (cond ((eq pv (caar plist)) ; check for member-ship
        (if (zerop
              (boole 1 (lsh 1 mv) (cdar plist)))
          (progn (setq elist (cdr elist)) (go NEXTelist))
           (return (cdar elist)))) ; YES this is the one
       (t (setq plist (cdr plist))
         (go NEXTplist)))
    ))) ; end of MVAREF-FIND
```