Towards a Facility for Lexically Scoped, Dynamic Mutual Recursion in Scheme Systems

by

John Franco and Daniel P. Friedman
Department of Computer Science
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 268

Towards a Facility for Lexically Scoped, Dynamic Mutual Recursion in Scheme Systems

by

John Franco and Daniel P. Friedman

January, 1989

This work was supported in part by the Air Force Office of Scientific Research grants numbered 84-0372 and 89-0186, and the National Science Foundation grant number CCR 87-02117.

Towards a Facility For Lexically Scoped, Dynamic Mutual Recursion in Scheme Systems

John Franco, Daniel P. Friedman

Department of Computer Science,

Indiana University, Bloomington Indiana 47405

January 24, 1989

Abstract

We propose a facility which allows unbounded associative structures, which we call ARRAYs, in SCHEME systems. An important application is the creation of unbounded vectors and arrays. Another application is as the underpinings of a global, dynamic letrec capability. A third application is the construction of memo-functions. Under this facility, ARRAY elements are allocated space individually and not until they are side-effected. Thus, ARRAYs can be sparse and waste little memory. In addition, the proposed facility removes some of the burdens of writing procedural specifications that are not relevant to functional specifications such as vector boundedness.

1 Introduction

The current trend toward object-oriented style raises the following problem: how does one represent a dynamic class of objects, each subject to common procedural actions, so that access to individual objects is immediate and all "live" objects are mutually knowledgeable. Such a problem might arise, for example, when modeling the interaction of processes of bounded lifetime.

Current Scheme systems force inelegant and even inadequate solutions to this problem. Vectors may be used to achieve immediate access but they are bounded structures and may not be able to accomodate dynamic object classes without wasting resources. On the other hand, lists do not allow immediate access. The `letrec` facility handles mutual recursion of arbitrarily many objects but cannot deal with objects that are created and destroyed on the fly. We propose to solve the problem by introducing a new class of structures called ARRAYs.

An ARRAY is an unbounded associative structure. Its use is similar to that of vectors in Scheme. An ARRAY is defined in a `let` or `letrec` using a special procedure `make-ARRAY` which is described in section 3.1. Its scope obeys the standard rules of scope for `let` statements. ARRAY elements are initially `NIL` and do not use memory resources until given a value by means of the special procedure `ARRAY-set!` which is described in section 3.2. The special procedure `ARRAY-ref`, described in section 3.3, is used to retrieve values of individual elements. ARRAYs are random access; that is, the average time to set or retrieve element values is bounded by a constant. ARRAY elements that are set to `NIL` cease to use memory resources.

In order to solve our problem, ARRAYs have a *semantic* component which is defined by means of an argument to `make-ARRAY`. This component is a procedure which specifies how individual objects, represented as individual ARRAY elements, interact with each other. The procedure is capable of creating or destroying other objects as well as passing knowledge to and from them.

The properties of ARRAYs make several generalizations of existing facilities available in a standard implementation. For example, memoization, unbounded vectors, sparse vectors and arrays, ucons cells, and dynamic letrec are all possible applications of ARRAYs. In particular, sparse arrays are implemented in exactly the same way dense matrices are with ARRAYs.

There are several possible implementations of ARRAYs depending on what scheme is used to locate elements. An implementation may use, for example, Hash Tables of Common Lisp [7], *Linear Hashing*, or *Spiral Storage* [4] in this regard. However, we shall make the assumption that large, fast memory will eventually be standard equipment. Then, virtual memory and caching will not be necessary to support a successful Scheme system. Without these, it makes sense to locate elements by hashing over all available space. That is, we propose to have a hash table for locating ARRAY elements such that individual cells in that hash table are allowed to occupy space anywhere in memory. This arrangement certainly gives the unbounded and random-access properties we need. It should also be more efficient than Common Lisp Hash Tables or Linear Hashing since copying or rehashing table elements during table-size changes is not necessary.

A possible problem with hashing over all available space is it complicates garbage collection. However, if there are few requests for large contiguous blocks of memory, then a simple and fast garbage collector can be employed. Furthermore, garbage collection can be done off-line and in hardware. Hence, we do not view garbage collection as a serious problem. Another weakness, in terms of stock hardware, is that a test is required to determine whether a cons cell is a hash table element just prior to allocating it. However, it seems possible to have this test done in hardware by a primitive processor, perhaps the one used for garbage collection. This processor would have the next cons cell ready when an allocation request needs to be handled.

This paper describes ARRAYs, their implementation, and some needed modifications to garbage collection. In section 2 we present an overview of the proposed facility. In section 3 the language of the facility is given. Section 4 contains a simplified implementation. Section 5 discusses garbage collection under the facility. Section 6 considers modifications to support requests for large data allocations.

2 Overview

The random access property of ARRAY elements is attained by hashing through special cons cells called *locators*. There is one unique locator for each side-effected ARRAY element. Locators are allowed to exist anywhere in available memory and the number of locators defined by the user is limited only by the size of memory. This has two benefits. First, under a standard, two-memory Scheme implementation, the average number of hashes to locate an ARRAY element is less than two. Second, there is no need for buckets or a secondary fixed hash table.

All locators belonging to the same ARRAY are in a list accessed through the ARRAY variable. Thus, it is possible to visit ARRAY elements sequentially. In the proposed implementation the sequence is in order of locator creation. Visiting elements in lexicographic order can be achieved if the list is kept sorted lexicographically.

Hashing is on an ordered list of key elements, called the *key-list*, which is obtained from a *key-structure* that must be specified by the user. A key-structure is any Scheme-definable object. Every key-list contains at least two ordered keys. At the root of the key-structure and the head of every key-list is an atom representing the ARRAY variable. Except for this atom, each key-list element may be any Scheme object. In the case of associative vectors and arrays the key-list is the key-structure and all keys except the first are atoms.

Associated with each ARRAY are two procedures called FUNC and PRED which define the hashing and matching of ARRAY elements, and a procedure NODE-DEF which defines the "semantics" of ARRAY elements. All three procedures are defined by the user when an ARRAY is created and cannot be changed once defined.

FUNC specifies how the key-list is to be assembled from a given key-structure. It takes a key-structure as input and returns a list of values that are to be hashed. A common definition is the identity function `(lambda (KS) KS)` which causes all members of the list `KS` to be hashed.

PRED defines the ARRAY element matching condition. It takes a list of keys `K` and a structure `S` to be matched as input and returns *true* if and only if certain ordered elements of `S` match the list of keys. A common definition is the equality function, `(lambda (K S) (equal? K S))` which returns *true* if and only if the list `S` is identical to the list `K`. That is, the values of all the key elements `K` must match the values of a given list `S`. Having PRED and FUNC greatly improves the generality of the proposed system. For example, as shown in section 3.3, the UCONS facility [6] can be completely written on top of the proposed facility.

NODE-DEF defines the "semantics" of ARRAYs. It allows ARRAY elements to be called as procedures and is useful in combinatorial solutions which involve extensive subproblem decomposition. In this role, ARRAY elements may be regarded as subproblems with "semantics" which describe the decomposition and reconstruction process and "value" which is the result of applying the "semantic" procedure. The "semantic" procedure can be used to insure that decomposition is performed only once per subproblem thereby greatly enhancing efficiency. The "loose" organization of ARRAY elements frees the programmer from the burden of producing a rigid structure in order to retain the results of subproblem computations. This rigid structure, which is common in Pascal-like implementations of Dynamic Programming and involves limiting structure size and enforcing interpretations to adjacent structure elements, is unnecessary from the point of view of the functional specification. Although all possible subproblems can be "generated" by our facility,

computational resources are expended only on the subproblems that are required for the solution. An example is given in the next section. A common definition is

```
(lambda (array-element)
  (ARRAY-ref array-name array-element))
```

That is, just the value of the ARRAY element `array-element` of ARRAY `array-name` is returned.

The time and space cost of maintaining this facility is proportional to the number of ARRAY elements in use. Thus, if no elements are in use there is no overhead penalty. The constant of proportionality of time overhead is low. The constant of proportionality of space overhead is low if ARRAY element values are moderately large procedures. Part of this cost is due to a slight modification that must be made to conventional garbage collection and compaction algorithms in order to accomodate the locators.

3 The Language of ARRAYs

The language of ARRAYs is analogous to and an extension of the language of vectors with some minor syntactic differences. ARRAYs are used to define families of locators. There are four operations for manipulating ARRAYs: definition, side-effecting, referencing, and invoking ARRAY elements.

3.1 ARRAY Definition

ARRAY definitions are created using a new procedure of three arguments called `make-ARRAY`. The first argument is NODE-DEF, the second argument is PRED, and the third argument is FUNC. For example, the following defines ARRAYs `ring` and `knap`:

```
(letrec ([ring (make-ARRAY
                 (lambda (array-element) (ARRAY-ref ring array-element))
                 (lambda (K S) (equal? K S))
                 (lambda (K) K))]
         [knap (make-ARRAY
                 (lambda (array-element) (ARRAY-ref knap array-element))
                 (lambda (K S) (equal? K S))
                 (lambda (K) K))])
  <body-of-letrec>).
```

3.2 Side-Effecting ARRAY Elements

The procedure `ARRAY-set!` which takes three arguments is used to create an ARRAY element, if necessary, and store a value in it. The first argument is an ARRAY, the next argument is the key structure, and the third argument is data. The value of the `ARRAY-set!` procedure is unspecified. An example of its use is

```
(ARRAY-set! ring '(1 2) 'abc)
```

which is equivalent to $ring(1,2) \leftarrow$ `'abc`. Row 1, column 2 of `ring` now exists physically, regardless of the status of other elements of `ring`.

3.3 Referencing ARRAY Elements

The `ARRAY-ref` procedure returns the value of an ARRAY element. An example of its use is the following:

```
(cons (ARRAY-ref ring '(1 2)) '(n))
```

which creates the list `(abc n)`. If the ARRAY element referenced by ARRAY-ref has never been side-effected then `NIL` is returned.

Earlier we mentioned that the ucons cell of [6] could be implemented efficiently using this facility. A ucons cell is a cons cell that can be accessed by looking at its contents. The ucons cell is useful in memoizing [3,5]. The code for the ucons operation is as follows:

```
(define ucons
  (letrec ([u-ARRAY
            (make-ARRAY
              (lambda (array-element) (ARRAY-ref u-ARRAY array-element))
              (lambda (K S)
                  (and (eq? (car K) (car S))
                       (eq? (cadr K) (cdr S))))
              (lambda (KS) (list (car KS) (cdr KS))))])
    (lambda (x y)
      (let ([c (cons x y)])
        (let ([b (ARRAY-ref u-ARRAY c)])
          (if (null? b)
              (begin (ARRAY-set! u-ARRAY c c) c)
              b))))))
```

For `u-ARRAY`, NODE-DEF returns the value of a specified `u-ARRAY` element, PRED returns *true* if and only if the two elements of the key-list match the car and cdr, respectively, of a specified cons cell, and FUNC produces a key-list which contains the car and cdr of a specified cons cell. Two arguments are passed to `u-ARRAY`: these are the car and cdr of a cons cell `c`. This cell is used as a key-structure to determine if the cell already exists as a ucons cell. If it does then `(ARRAY-ref`

...) returns a non-null value to b. Otherwise the expression (ARRAY-set! u-ARRAY c c) causes the data in the cons cell to be shared as the key-structure and its value to be returned by procedure ucons. From then on, the cons cell may be referenced directly by the contents of its car and cdr via an (ARRAY-ref u-ARRAY ...). This is precisely the property needed by ucons cells.

3.4 Invoking ARRAY Elements

As an example of the invocation of ARRAY elements consider the Partition Problem which is defined as follows: given a set $A = \{a_1, a_2, ..., a_n\}$ of objects, a weighting function $w : A \rightarrow N^+$, and an integer K, does there exist a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} w(a) = K?$$

This problem may be solved by subproblem decomposition. For example, suppose $K = 6$ and the list $W = \{1133\}$ represents the weights of the objects of a given A, we may decompose this into the two subproblems $W_1 = \{133\}$, $K = 6$, and $W_2 = \{133\}$, $K = 5$ corresponding to $a_1 \notin A'$ and $a_1 \in A'$, respectively. The answer to the original problem is "yes" if and only if the answer to either of the subproblems is "yes". In this example, the answer to the first subproblem is "yes" and the answer to the second subproblem is "no" so the answer to the original problem is "yes". Solutions to the subproblems may be found by further decomposition until primitive subproblems are reached.

A solution to the Partition problem using this decomposition is

```
(define partition
  (lambda (K W)
    (letrec
      ([part
         (make-ARRAY
           (lambda (K^ W^)
             (let ([y (ARRAY-ref part K^)])
               (cond [(not (null? y)) y]
                     [(eq? K^ 0) #T]
                     [(< K^ 0) #F]
                     [(null? W^) #F]
                     [else (ARRAY-set!
                             part
                             K^
                             (or (part (- K^ (car W^)) (cdr W^))
                                 (part K^ (cdr W^))))]))))
           (lambda (K S) (equal? K S))
           (lambda (K) K))])
      (part K W))))
```

where the list `W` is the list of object weights. The form of this solution is similar to the solutions to certain graph problems in [2] using extend-syntax. The extend-syntax macro expansion facility cannot be used to solve the Partition problem efficiently, however, since the subproblems, which are analogous to vertices in the graph problems, are *not* known at compile-time and are uncovered only during run-time. The proposed facility makes these subproblems known to all subproblems when they are created. In other words, creation of the ARRAY `part` has the same effect as defining its elements using a form of *dynamic letrec*.

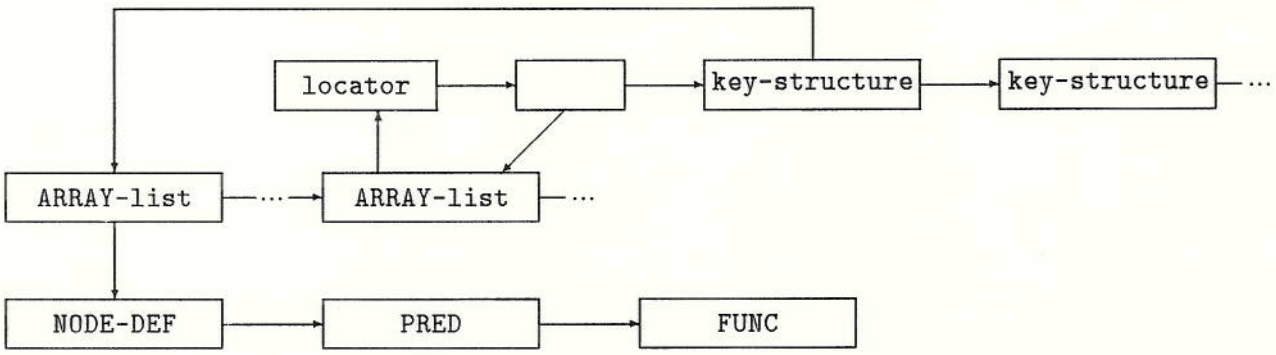
As an example, we consider the call `(partition 6 '(1 1 3 3))`. Six locators are created by this call. We number them from 1 to 6 corresponding to ARRAY elements having keys from 1 to 6. Locators are given values in the following order: 1,4,2,5 are given the value *false*, then 3,6 are given the value *true*. Locator 5 gets the value *false* because it is created as a result of the call `(part 5 '(1 3 3))` and there is no subset of {133} which sums to 5. Similar statements can be made to account for the *false* values given to locators 1,2, and 4. Because recomputation of the values of locators 1,2, and 5 is unnecessary, the number of procedure calls is reduced by a factor of two.

4 A Simplified Implementation

We describe an implementation that is efficient but does not handle requests for large blocks of contiguous memory. In a later section we outline a method for handling large memory requests.

The necessary structures are locators, key-structures, key-lists, and ARRAY-lists. Key-structures and key-lists have already been described. We need to supply more information about locators. The address of a locator is obtained by hashing on a key-structure. A locator may also be reached from an ARRAY variable through an ARRAY-list. No locator can be reached through an inaccessible ARRAY. The car of a locator is its value and is an ARRAY element. The cdr of a locator is a cons cell with a pointer in its car and a key-structure in its cdr. The pointer is used during garbage collection and will be described later. The extent of a locator is inherited from the extent of its associated ARRAY variable. The addresses of locators, key-structures, and key-lists are hidden from the user.

The ARRAY-list facilitates garbage collection, sequential visitation of ARRAY elements, and allows access to NODE-DEF, PRED and FUNC. It is a singly-linked list pointed to by an ARRAY variable and is hidden from the user. The first cell of the ARRAY-list points to a list containing NODE-DEF, PRED and FUNC. The first cell and the list of procedures are created by `make-ARRAY`. Each element in the ARRAY-list except the first points to a locator which is associated with the ARRAY. The car of the cdr of that locator points back to the ARRAY-list element (this facilitates garbage collection). As each new associated locator is created, a cell pointing to it is added to the ARRAY-list. No locator can be created unless it can be referenced from an existing ARRAY. The relationship between locators, the ARRAY-list, the key-list, and the ARRAY variable are given in the figure below. From this we see that a locator is marked during garbage collection if and only if the associated ARRAY is marked.



Allocating memory for a locator that does not exist or referencing an existing locator is simple. Let KEY denote a key-structure. The result of FUNC(KEY) is hashed to address C^*. If C^* is not a locator and in use then FUNC(KEY) is rehashed until either C^* is a locator or is not in use. Suppose C^* is a locator and let $C^*.key$ be the address of the key-structure associated with locator C^*. If PRED(FUNC(KEY),FUNC($C^*.key$)) is *false* then FUNC(KEY) is rehashed. If C^* is a locator and PRED(FUNC(KEY),FUNC($C^*.key$)) is *true* then C^* is returned. Otherwise, C^* is not in use and it is marked as a locator, a cons cell is allocated to the cdr of C^* and C^* is returned. Then an ARRAY-list element is created and the car of the cons cell is set to point to that element. The cdr of the cons cell is given the address of KEY. For `ARRAY-set!` with a non-`NIL` value-argument the value field of C^* is altered, for `ARRAY-ref` it is returned. If `NIL` is given as a value-argument to `ARRAY-set!` then, if the associated locator is found to exist, the reference to it from the ARRAY-list is broken. Thus, the locator and any non-shared structures it references will be garbage collected. Furthermore, during garbage collection, the ARRAY-list cell with the broken link is removed from the ARRAY-list.

5 Garbage Collection and Compaction

Garbage collection and compaction are accomplished using two memories of equal size. One memory is referred to as active and the other inactive. Except for locators, allocations come from the active memory until it is used up. Then, except for locators, the good cells are collected and compacted in the inactive memory. We use a pointer, called BP, to locate the destinations of the good cells in inactive memory. At the outset of garbage collection, BP points to the lower boundary of inactive memory. Upon completion of compaction, the two memories switch roles.

Locators may appear in either memory at any time. They are not moved unless a key value is changed due to compaction (this might happen, for instance, if the key value is an address). In the remainder of this section we detail the garbage collection and compaction mechanism. We assume that all memory requests are for cons cells.

First, all active structures are traversed and cells marked. Except for locators, the contents of each active cell are copied to the cell pointed to by BP and BP is advanced by one cell. If the next cell is a marked locator then BP is repeatedly advanced by one cell until it does not point to a marked locator. A forwarding pointer to the new cell position is left in the old cell position. The forwarding pointer is used later to reset pointers to the moved cell.

When copying is complete, all used cells in inactive memory are checked for pointers into active memory. Each pointer in a used cell of inactive memory that references a cell in active memory is replaced by the forwarding pointer in the referenced cell.

Next, all active ARRAY-lists are traversed again. The key-lists of their locators are checked to see if the hashing address has changed. This is accomplished by hashing until either an empty address or the locator itself is found (all unmarked locators are considered free space). If it's the locator then nothing happens. If it's an empty address the locator is copied to that address and the associated ARRAY-list element is updated (access is through the car of the cdr of the locator). If, during traversal, an ARRAY-list element is found not to reference a locator then it is removed from the ARRAY-list.

6 Unbounded ARRAYs Under Requests For Large Allocations

For an efficient implementation of unbounded arrays under requests for large allocations, it is sufficient to manage multiple blocks of free memory efficiently. Assume two-memory garbage collection and compaction as above, allocations are taken from the active memory until no sufficient space is available, and locators are allowed to exist anywhere in memory. The space between locators in active memory is free space that is available. We call each such space a free-block.

Requests for memory could be satisfied sequentially as in the previous section. This method for handling block requests would be satisfactory if block sizes are usually much less than the size of total memory divided by the number of locators. In this case, if the current-free-block is not big enough to accomodate the request, the next one will most likely be. Thus, the time overhead is small. If block sizes are too large, however, a long walk through many free-blocks might be required before a free-block big enough to service the request is found. In this case, at the cost of a factor of $O(ln(n))$, where n is the number of locators, the following method for managing free memory is preferred.

The problem of choosing a free-block for allocation may be solved by maintaining a heap of *block-descriptors* or *descriptors*, one for each free-block and containing a pointer to that free-block, organized on free-block size. It is well known that unit time is required to find the largest element of a heap and only $ln(n)$ time is required to re-heapify if one or two element sizes are changed [1]. Define the *current-block* to be the free-block from which memory requests are satisfied. All requests are taken from the current-block until it is too small to handle one. Then the largest free-block is checked to see if it can handle the request. If so, it becomes the current-block and allocation continues. If not, a garbage collection and compaction are performed.

Locator creation generally splits a free-memory-block into a pair of much smaller blocks. Then one descriptor must be updated, one must be created and the descriptor heap must be updated. The problem of finding the free-block in which the locator is created is solved using a Trie [1]. The leaves of the Trie point to block-descriptors. To identify the free-block associated with an arbitrary

address, one walks the Trie, from root to leaf, using the address bits to decide direction at each node. The block-descriptor pointed to by the leaf contains the pointer to the free-block. Creation of the new locator affects the dimensions of the current-block; therefore, the current-block descriptor must be updated. If the locator is not at the boundary of the current-block then a new descriptor is created, sized, and set to point to $C^* - 1$. The descriptor heap and Trie may need to be updated possibly employing re-heapification.

Garbage collection and compaction are complicated by the fact that a locator must be moved if it occupies space that must be taken by a contiguous block of cells during compaction. In this case a secondary list of locators-to-be-moved is constructed in inactive memory during compaction. After compaction the locators in this list are rehashed to their proper addresses.

7 Conclusions

We have presented a facility which allows unbounded associative stuctures in Scheme systems. Some of the benefits of the proposed facility are 1) unbounded vectors and arrays, 2) memoization, 3) ucons cells, 4) dynamic letrec. The facility allows removal of some procedural specifications that are unnecessary in stating the functional specification of many programs. In particular, unbounded vectors and arrays eliminate the need to have boundedness conditions in the specification. The facility also allows certain solutions, based on subproblem decomposition and reconstruction, to be realized without requiring a rigid data structure. Thus, Dynamic Programs can be implemented at full asymptotic efficiency up to a constant factor with little thought about the data structures used.

We have provided only enough programming examples to illustrate the use of the facility. Many other examples were omitted to save space. Programs in which interacting objects are born and die could benefit from this facility.

References

[1] **A. V. Aho, J. E. Hopcroft, and J. D. Ullman,** *The Design and Analysis of Computer Algorithms,* Addison-Wesley (1974).

[2] **J. Franco, and D. P. Friedman,** "Creating efficient programs by exchanging data for procedures," to appear in *Computer Languages.*

[3] **J. Hughes,** "Lazy memo-functions," in *Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science #201, ed. Jean-Pierre Jouannaud, Springer Verlag (1985), pp. 129-146.

[4] **P. Larson,** "Dynamic Hash Tables," *C.ACM* **31**, No. 4 (1988), pp. 446-457.

[5] **D. Michie,** "'Memo' functions and machine learning," in *Nature*, #218 (1968), pp. 19-22.

[6] **J. M. Spitzen, K. N. Levitt, and L. Robinson,** "An example of hierarchical design and proof," *C.ACM* **21**, No. 12 (1978), pp. 1064-1075.

[7] **G. L. Steele, Jr.,** *Common Lisp,* Digital Press (1984).