数理解析研究所講究録
第436巻 1981年 117-131

Yet Another Environment-Retention Strategy

Hiroshi TOSHIMA

Department of Management Sciences
Faculty of Commerce
Otaru University of Commerce

ABSTRACT: The simple and usual stack operations are generalized to handle the more complex data structure than the vector in the manner like the stack. As a consequence of this generalization, some environment-retention strategy can be described in terms of the generalized stack operations. This strategy enables the unnecessary local environments allocated in the heap to be instantaneously collected without recourse to garbage collections but the binding environments to be retained so as to evaluate correctly the Lisp programs involving the functional values. As an "obiterdictum", the method of the instantaneous collection of the list created by the function evlis is briefly stated.

Keywords and phrases: Lisp, Stack, A-list, Environment, Tree Structure, Functional Argument, Functional Value, Retention, Garbage Collection.

1. Introduction

One of the most characteristic features in Lisp programming is that one can return a function as the result of some computation. In fact, this feature is not supported by most programming languages, e.g. ALGOL 60, FORTRAN, etc. Following Allen's terminology [1], we shall call a returned function a functional value[*1] as distinguished from a functional argument.

*1 Also called upward funarg.

The functional value, however, bears a hard nut to crack when one plans to implement Lisp system. Consider the evaluation of the following form (cited from Allen [1]).

f [cons [A ; (B C)] ; g[function [car] ; function [cdr]]],

where

f [p ; q] = q [p],
g [u ; v] = function [λ [[x] ; u [v [x]]]] .

It is easily seen that the variable-value pairs corresponding to the form g [function [car] ; function [cdr]] must be retained after the evaluation of this form is over. LISP 1.5 interpreter works completely well for the evaluation of the above form. As is well-known, LISP 1.5 interpreter is characterized by an association list (abbreviated hereafter as A-list) which represents an environment. Lambda binding adds the new variable-value pairs to the A-list. After the evaluation of the specific form is over, the corresponding variable-value pairs are retained against garbage collection if the pointer to that pairs still exists. Thus, in LISP 1.5 interpreter, the A-list retains the binding environment as long as it is needed and then it is garbage collected when it is no longer needed. In this sense, LISP 1.5 interpreter makes it easy to retain the arbitrary environment. But the A-list has two drawbacks when the retention of the environment is not necessary. First, the unnecessary variable-value pairs occupy the memory. Second, this occupation results in costly garbage collection. Due to this expensiveness of the A-list in both time and space, LISP 1.6 interpreter

and MACLISP interpreter, etc. adopt the stack to store the variable-value pairs. As the stack behaves in last-in-first-out fashion, the unnecessary variable-value pairs are automatically deleted from the memory, so that the memory management by the stack is less expensive than the one by garbage collection. More specifically, in LISP 1.6-like interpreter, the current bindings are stored on the value cells and the previous bindings are saved on the stack. The binding environment can promptly recovered by suitable replacing of the contents of the value cells by the contents of the stack. But there is one serious problem in LISP 1.6-like interpreter. Unfortunately LISP 1.6-like interpreter fails for the evaluation of the above form. Note that LISP 1.6-like interpreter leads to premature popping up of the binding environment, i.e., the necessary variable-value pairs have been lost when the form u[v[x]] is evaluated. In view of such a situation, Moses [7] pointed out that a tree structure must be developed from what was originally a linear stack and this tree structure must be backed-up and backed-down upon entering the returned function. It should be noted that a tree structure is usually linked only in one direction. Thus, to back-down a tree structure can not be achieved without some inefficiency. As the A-list is a tree, this is again the inefficiency of a tree structure. Summing up the above discussion, to retain the environment, one must adopt the A-list or, generally speaking, a tree structure. To keep the spatial and temporal efficiency, one must adopt the stack. How can one reconcile these two?

In a response to Moses [7], Sandewall[8] gave some solution to the above problem. His idea is essentially same as the one adopted in INTERLISP*1 [9]. The function-construct evaluates to the usual funarg-triple, the third element of which is not the A-list but the array involving variables

*1 Old version.

freely used by the second element of funarg-triple, namely a function, and their values at the time the function-construct is evaluated. When the funarg-triple is applied to its arguments, each element of the array is pointed to indirectly from the stack(Sandewall) or the stack is made look like it has a patch of the array (INTERLISP). These solutions use the stack modified by the data structure in the heap which is copied from the entries of the stack. Recently, McDermott [6] proposed a new solution hunting down this idea of "the stack enhanced by the heap". He allocates variable-value pairs on the stack, then moves them to the heap if necessary. He also passes the actual parameters to the function via the stack. As a result of these, he saves dramatically on garbage collections. If the function-construct is evaluated to the lambda expression[*1] in the process of the application of which to the arguments the free variables are bound to the proper values by the usual lambda binding mechanism, this method is called the optional freeze and is adopted by POP-2 and HLISP [5]. Then the optional freeze dispenses with indirect addressing from the stack or patching of the stack, so that the same data structure in the heap is utilized every time the evaluation of the form whose function part is the above lambda expression is done. So the optional freeze can not fully reflect the effect of the functions set and setq on the environment but the implementation of the optional freeze is relatively easy.

Bobrow and Wegbreit [3] proposed the stack implementation of multiple environments (spaghetti stacks) and showed that the functional value can be handled properly as a special case of multiple environments. Clearly, their method needs not any data structure in the heap. On the contrary, there are

*1 In fact, such a lambda expression can be easily constructed as a good exercise to the Lisp buff!

the methods without the stack. Gleenblatt [4] proposed the following solution. If the value cells and a tree structure are used to represent an environment, only one branch of a tree structure is active at any time. The nodes along the path starting from the root of the tree to the last node of that active branch are assumed to be marked "down" and all other nodes marked "up". To back-up and back-down a tree structure from the current environment to the binding environment, traverse first a tree structure from the last node of the binding environment until the node marked "down" is found and turn each pointers around on the way and remember that node. Then traverse next a tree structure from the last node of the current environment to the last node of the binding environment via the remembered node and turn each pointer around on the way from the remembered node. Traversing a tree structure must also accompany swapping of the contents between the value cells and the nodes. This establishes the binding environment on the value cells. Baker [2] developed the more elegant solution by pondering the above scheme. His method called "rerooting" depends on continuous turning around of the pointers so as to make the current environment a new root. In this method, the path from the binding environment to the current environment is always easily traversed and the marks "down" and "up" can be dispensed with. Furthermore, this rerooting leaves the form assoc[v ; a] invariant for all variables v and all environments a serendipitiously. But, as Baker pointed out, this rerooting model is not necessarily most efficient since a tree structure is retained in the heap rather than on the stack, thus resulting in costly reclamations.

In this paper we propose a new model which works completely well for the Lisp programs involving the functional values. Our model is similar to a tree structure, the model retaining the necessary environment as long

as it is needed and it is garbage collected when it is no longer needed, or it is similar to the stack, the model showing stack-like behavior so that the number of costly reclamations may be detered as small as possible. Essentially, our model does depend on the generalized stack operations. Then, section 2 describes the generalized stack operations. Section 3 discusses the data structure of an environment and the operations to handle the data structure of an environment as an application of the generalized stack operations. Section 4 suggests some extentions and describes the concluding remarks.

2. The Generalized Stack Operations

As stated in the previous section, the A-list is inefficient in the memory management. Instead, if the (linear) stack is used to store the variable-value pairs, only thing to do is suitable managing of the stack-top pointer, which is a rather simple operation. In fact, the stack-top pointer needs to be modified only by one upon pushing or poping of one item. But the (linear) stack, i.e., the vector, is not so relevant as the vehicles to express the more complex data sturcture. So the efficiency and the power of expression are not capable of coexistence for the usual (linear) stack. In what follows, we shall somewhat extend the stack operations so as to handle the more complex data structure than the vector in the fashion akin to the stack and, as in the previous section, use the Lisp terminology freely. The global pointers g1 and g2 shall represent the stack-top pointer and the free memory pointer respectively. Generally the initial value of g1 is NIL. First, consider the following pushing operation.

push[x : f] = [null[f] ⟶ g1 := cons2[g1 ; x] ;

```
                T ⟶ g1 := f [ g1 ; x ] ],
cons2[ x ; y ] = prog [ [ z ] ;
                [ null [ g2 ] ⟶ reclaim [ ] ] ;
                  z := g2 ;
                  g2 := cdr [ g2 ] ;
                  rplaca [ z ; x ] ;
                  return[ rplacd [ z ; y ] ] ],
```

where x is the data structure to be pushed down and f is a functional argument which allows the arbitrary operations on g1 and x. If the vector of the cells is preserved somewhere in advance and g1 initially points to the first cell of the vector, the function cons3 defined as follows makes the function push behave the usual pushing operation except for the redundant linking information.

```
cons3[ x ; y ] = prog [ [ z ] ;
                [ eq [ g1 ; LIM ] ⟶ return [ ERROR ] ] ;
                  z := g1 + 1 ;
                  rplaca [ z ; x ] ;
                  return [ rplacd [ z ; y ] ] ],
```

where LIM is the length of the vector. In this case, the function push must be called as push [x ; function [cons3]]. In the general case, the function push is called as push [x ; NIL]. Note that the free memory pointed to by g2 is not required to be contiguous but it may consist of the cells chained by each cdr part, namely, it may be a list. Remember that the arbitrary data structure can be constructed by using the function push and suitable

setting of gl.

Second, consider the following popping operation.

```
pop [ x ] = prog [ [ ] ;
            g0 := 0 ;
           [ null [ x ] ⟶ go [ L2 ] ] ;
  L1        g0 := g0 + 1 ;
           [ null [ eval [ car [ x ] ] ] ⟶ return [ gl := car [ gl ] ] ;
            null [ cdr [ x ] ] ⟶ go [ L2 ] ] ;
            x := cdr [ x ] ;
            go [ L1 ] ;
  L2        g0 := 0 ;
            rplacd [ cadr [ gl ] ; g2 ] ;
            g2 := gl ;
            return [ gl := car [ gl ] ] ],
```

where x is a list of the forms and g0 is a global variable. If the value of some element of x is NIL, returning of the memory occupied by the unnecessary data structure (i.e., garbage) to the free memory pool is skipped and the cause of this skip is accessible by g0 whose value indicates the position of the cause in x. If any element of x is not NIL or x is NIL, garbage are returned to the free memory pool. In both cases, gl is suitably modified. It should be noted that the data structure to be popped off is expected to be non-branched and, in general, to have a pointer to the last cell of the data structure at cadr part of the data structure, i.e., cadr [gl] must show the location of the last cell. If the function pop is implied to apply to the data structure constructed by the function push, the data structure to

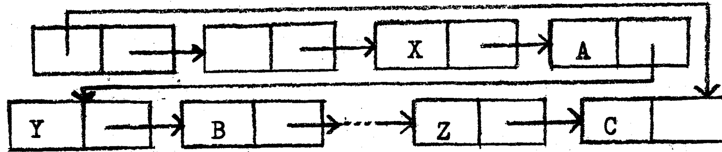
be pushed down must have the same structure. For the generalized popping operation, the function pop is called as pop[NIL], which causes returning of the memory. But, for more general case, the function pop must be called as pop[(c1 c2 ..... cn)] where ci's are the possible causes of the above skip.

We expect that the Lisp system not only has the functions push and pop similar to the above as the primitives but also these operations can be executed by the hardware instructions. Recently, many computers including the micrcomputers become to have the instructions relating to the stack operations. They are, however, too simple to handle the complex data structure in view of above explanation. Our functions push and pop try to generalize the simple stack operations. In fact, if the vector of the cells is assumed to exist, push[x ; function[cons3]] and pop[(NIL)] represent the usual simple pop and push operations. But these are the extreme cases in our stack operations. The functions push and pop in general do not assume the vector but permit the data structure to take the form of the arbitrary list structure. The unnecessary part of the list structure is retruned by the function pop just as the simple stack operation achieves the efficient memory management. In the conventional Lisp system, the unnecessary variable-value pairs in the A-list are garbage collected intermittently. In contrast to this conventional Lisp system, our function pop dispenses with garbage collections as to an environment completely at least for the case of pop [NIL]. This will be explained in the next section more detailedly.

3. The Data Structure of an Environment and the Operations upon an Environment

The following sort of the data structures is called the local envi-

ronment where X, Y,..., Z are variables and A, B,..., C are values.



An environment consists of the cells whose cdr parts point to the local environments and whose car parts are chained in one direction. The local environment is created by the function pairlis upon lambda binding and collected by the function apply upon the completion of the evaluation of the corresponding form if it is no longer needed. Note that the variable-value pair on the local environment is not an usual dotted pair but is a non-branched list. So the local environment is nothing other than the non-branched A-list. The car part of the first cell on the local environment points to the last cell on the local environment. Thanks to the existence of this pointer to the last cell the collection time of the local environment is constant regardless of the length of the local environment. Note that the non-branchedness of the local environment is beneficial to returning of the cells constituting the local environment to the free memory pool. In fact, the procedure of the collection of the local environment and the cell pointed to by gl is simply described by pop[NIL] where gl is the global pointer to one of the cells on an environment. This explains why the cells on an environment are car-chained each other, not cdr-chained.

Whether the specific local environment may be collected or not is controlled by the global flag, say, g4 and the entry in the car part of the second cell on the local environment. If g4 is NIL, the current local environment must be retained unconditionally and otherwise the car part of the second cell on the current local environment must be checked. If it is NIL, the retention of the current local environment is necessary and other-

wise the retention is not necessary. The suitable control of g4 and the car part of the second cell on the local environment guarantees the versatile environment-retention strategy for the functional value. For example, the car part of the second cell is found to be NIL in the process of the collection of the local environment, g4 is set to NIL because all the local environmrnts accessible from that local environment must be retained. Thus, the full process of collection of the local environment is described by pop[(g4 caddr[gl])]. If g0 = 1 then there is nothing to do more and if g0 = 2 then g4 must be set to NIL. The search of an environment begins from the current local environment pointed to by gl indirectly. If the search of the current local environment is unsuccessful, the downward local environments are searched until the value of variable is found. When the value of variable is found outside the current local environment, it might possibly save the search time afterward to add the grasped variable-value pair to the current local environment as the new entry. To follow the change of the environment by the functions set and setq, the new entry to be added must consist of the pointer to the original variable-value pair, i.e., the new entry must show the indirect addressing.

The retained local environments will eventually be reclaimed by garbage collections. Thus our system described above recovers the memory as efficiently as LISP 1.6 interperter does when no functional values are found in the Lisp program. As mentioned in section 1, LISP 1.6 interpreter can not evaluate correctly the program involving the functional values. But our system works completely well for such a program as LISP 1.5 interpreter does. In this case, garbage are not so efficiently collected as in LISP 1.6 interperter. The part of the memory used for the local environments is retained and eventually reclaimed by usual garbage collections. In this

sense, our system shows intermediary behavior between LISP 1.5 interpreter and LISP 1.6 interpreter. If all the car parts of the second cells in the local environment are set to NIL, then an environment takes the similar form as the A-list in LISP 1.5 interpreter. On the contrary, if all the car parts of the second cells in the local environments are set to non-NIL, then our system can not also evaluate correctly the program involving the functional values as in LISP 1.6 interpreter.

To digress: As our system does not use the stack, the evaluation of the actual arguments given to the function is performed by the usual function evlis. The cells are consumed by the function evlis each time the form is evaluated. But those cells become garbage as soon as lambda binding of the evaluation of the form is over. Then the huge consumption of the cells by the function evlis is a fatal drawback of the function evlis. It is one of the reasons for the frequent garbage collections, e.g. in the case of deep recursion. The gain from the stack-like action of an environment mentioned above might be offset by this consumption if we let the function evlis be as it is. Thus, the instantaneous collection of those cells might contribute the promotion of the spatial efficiency of our system. We shall use two global variables g20 and g21 whose initial values are NIL's for the instantaneous collection of those cells. Variable g20 will point to the last cell on the list created by the function evlis and variable g21 will point to the first cell on the list created by the function evlis. Now the value of g20 is set by the function evlis as follows.

```
evlis[m] = [null[m] ⟶ prog[[] ;
                          [null[g2] ⟶ reclaim[]] ;
                          g20 := g2] ;
```

```
T ⟶ cons2 [eval [car [m ] ] ; evlis [ cdr [m ] ] ] ].
```

To set the value of g21 and to collect the list created by the function evlis, we must replace in the function eval the form apply [car [e] ; evlis [cdr [e]]] by the form

```
prog [ [z] ;
  z := apply [findfun  [ car [ e ] ] ; g21 := evlis [ cdr [ e ] ] ] ;
 [null [ g20 ] ⟶ return [z ] ] ;
  rplacd [g20 ; g20 ] ;
  g2 := g21 ;
  g20 := NIL ;
  g21 := NIL ;
  return [ z ] ],
```

where e is an argument of the function eval and the function findfun ultimately evaluates the function part of the form e to find the function. When car [e] is the lambda expression, the list created by the function evlis can be collected immediately after lambda binding occurs in the function apply by the similar technique as mentioned above.

4. Suggestions on Some Extentions and Concluding Remarks

It is not difficult to extend our system so as to have a primitive retain [ ] by which the car part of the second cell in the current local environment is set to NIL to make all the local environments that can be reached from the current local environment be retained. The coroutines and backtracking might be brought to realization if the primitive retain [ ] is

properly used. More complex operations are also possible. The specific local environment can be retained by controlling the value of g4. In fact, after invocating the primitive retain [ ] the local environment will be retained if g4 is not set to NIL in the function apply.

So far deep binding was assumed in our model. If one wants to adopt shallow binding strategy, the Baker's rerooting algorithm [2] can be embodied in our scheme. In original Baker's model all the pointers linking each variable-value pair must be turned around. Rerooting in our model, however, requires only the pointers linking each cell on an environment to be turned around. This will cut down the rerooting overhead time substantially. Furthermore, our model with rerooting will achieve the more efficient memory utilization than the original Baker's model since the unnecessary local environments are collected instantaneously in our model. Thus, our model with rerooting will be less expensive in both time and space than the original Baker's model.

We have presented the model with an environment consisting of the local environments involving the non-branched A-lists and explained that our model shows the intermediary behavior between the A-list and the stack. Thus, our model works completely well for the Lisp program involving the functional values with the sufficient efficiency.

References

[1] Allen, John, Anatomy of LISP, McGraw-Hill, New York, 1978.

[2] Baker, Jr., Henry G., "Shallow Binding in Lisp 1.5", CACM, Vol. 21, No. 7 (July 1978), pp. 565-569.

[3] Bobrow, Daniel and Wegbreit, Ben, "A Model and Stack Implementation of Multiple Environments", CACM, Vol. 16, No. 10 (October 1973), pp. 591-603.

[4] Greenblatt, R., "The LISP Machine", Working Paper 79, M.I.T. A.I. Lab., Cambridge, Mass., November 1974.

[5] Kanada, Yasumasa, "Problems in the Impementation of HLISP, and Transporting REDUCE-2", Proceedings of Programming Symposium at Tateshina, Information Processing Society of Japan, 1974, pp. 14-20.

[6] McDermott, Drew, V., "An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-scoped LISP", Conference Record of 1980 LISP Conference, 1980, pp. 154-162.

[7] Moses, Joel, "The Function of FUNCTION in LISP or Why the FUNARG Problem Should be Called the Environment Problem", SIGSAM Bulletin, No. 15 (July 1970), pp. 13-27.

[8] Sandewall, Erik, "A Proposed Solution to the FUNARG Problem", SIGSAM Bulletin, No. 17 (January 1971), pp. 29-42.

[9] Teitelman, Warren, INTERLISP Reference Manual, Xerox Palo Alto Reseach Center, Palo Alto, California, 1974.