MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Working Paper 138 January, 1977

Shallow Binding in LISP 1.5

by
Henry G. Baker, Jr.

Shallow binding is a scheme which allows the value of a variable to be accessed in a bounded amount of computation. An elegant model for shallow binding in LISP 1.5 is presented in which context-switching is an environment structure transformation called "re-rooting". Re-rooting is completely general and reversible, and is *optional* in the sense that a LISP 1.5 interpreter will operate correctly whether or not re-rooting is invoked on every context change. Since re-rooting leaves `(ASSOC X A)` invariant, for all variables X and all environments A, the programmer can have access to a re-rooting primitive, `(SHALLOW)`, which gives him dynamic control over whether accesses are shallow or deep, and which effects only the speed of execution of a program, not its semantics. So long as re-rooting is an indivisible operation, multiple processes can be active in the same environment structure. The re-rooting scheme is compared to a *cache* scheme for shallow binding and the two are found to be compatible. Finally, the concept of re-rooting is shown not to depend upon LISP's choice of dynamic instead of lexical binding for free variables; hence it can be used in an Algol interpreter, for example.

Key Words and Phrases: LISP 1.5, environment structures, FUNARGs, shallow and deep binding, multiprogramming, cache.

CR Categories: 4.13, 4.22, 4.32

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0522.

A severe problem in LISP 1.5 [1] systems is the amount of time needed to access the value of a variable. This problem is compounded in LISP and APL by the "fluid" or "dynamic" binding of free variables; it is not so bad in Algol and PL/I, where free variables are "lexically" bound. Dynamic variable binding requires that free variables be looked up in the environment of the caller (dynamically embracing block) rather than that of the statically embracing block. This decision leads to environment trees which are "tall and thin" rather than "short and bushy". Since the length of time required to access the binding of a variable is proportional to the distance in the tree from the current environment to the place where that variable is bound, this time can be quite large with tall environment trees. For example, accessing a "global" variable at the bottom of a deep recursion can require time proportional to the depth of the recursion.

The MACLISP interpreter [2] solves this problem through a scheme called "shallow binding". In this scheme, variables have "value cells" as well as bindings in the environment. The scheme endeavors to always keep the bindings associated with the current environment in the "value cells" so that they can be accessed without any search. Whenever a context change occurs, such as when calling or returning from a function, these "value cells" must be changed or restored. Since the changes in the environment structure mirror those in the return-point stack, MACLISP implements shallow binding by saving old bindings on a stack when calling and popping them off when returning. Thus, there is a tradeoff between the time needed to access the binding of a variable and the time needed to switch contexts. In so-called deep binding systems like LISP 1.5, the access time is unbounded but the context switching time is constant. In MACLISP, access time is constant, while context switching time is unbounded.

The scheme used in MACLISP does not qualify as a model for shallow-binding in LISP 1.5 because it does not handle function-producing functions, i.e. upward FUNARGs, correctly [3]. No *stack* environment has that capability because conforming to stack discipline can lead to premature unbinding of variables. Our model for shallow binding keeps a *tree* environment and hence is capable of handling full FUNARGs.

In our model, we assign to each variable a *value cell*, which contains the default value of that variable, i.e. the value to be used if the variable is not bound in the in the path from the current environment to the root of the environment tree. The algorithm for accessing a variable is: 1) search the environment tree from the current environment to the root and if the variable is found, use the associated value; else 2) use the value in the value cell. (This scheme is used in several LISP 1.5 systems where the default value is the *global* or *top level* value of the variable). So far, we have described a conventional deep binding interpreter. In order to convert this system into a *shallow* binding system, we need only make sure that the distance from the current environment to the root in the environment tree is *short*; i.e. *make sure that the current environment is always the root*. In this case, the variable search is always trivial and is eliminated.

The way that we do this is to *re-root* the environment tree at each context-switch, so that the context to be switched to becomes the new root. Consider the situation in which a function of one parameter, X, is called from the null environment. Suppose (see Fig. 1) that X is bound to 5 in the null environment e_0 (i.e. X has 5 in its value cell), and that X is to be bound to 6 in the body of the function. The function call creates a new environment e_1 which pairs X with 6 and has e_0 as its parent environment. A deep binding interpreter would set the current environment pointer to e_1 and all variable accesses would be done through e_1, thus lengthening the access time to all variables but X. A shallow binding call, on the other hand, transforms the environment tree in the following manner: 1) e_0 is changed to pair X with 5 and have e_1 as its parent; 2) e_1 is changed to a null environment; and 3) the value cell of X is set to 6. In this way, *all* variable access are shallow inside the function. Upon return from the function, the environment is re-rooted back to e_0.

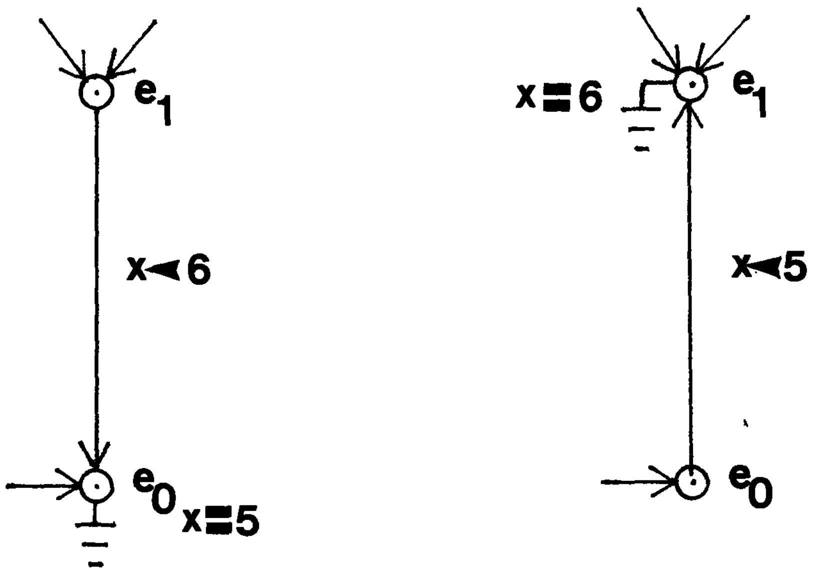

Figure 1. Environment transformations

We outline a proof that `(ASSOC X A)` is left invariant under the re-rooting transformation. There are only four basic cases: 1) `(ASSOC X e0)`; 2) `(ASSOC y e0)`; 3) `(ASSOC X e1)`; and 4) `(ASSOC y e1)`; where y is any variable other than X. In each case, the reader can check that the expression is left invariant. By induction on the length of the search path, `(ASSOC X A)` and `(ASSOC y A)` are left invariant for all other environments A. Finally, if the root can be moved by one step in the tree, it can be moved to any node in the tree by iterating the one step procedure. Thus, we have proved that re-rooting to any node in the tree is possible while preserving `(ASSOC X A)`, for all variables X and all environments A.

We present a modified ASSOC function which gets a variable binding from its value cell and a modified APPLY function which re-roots the environment tree whenever a context switch is made, i.e. when a function is called or returned from. RE-ROOT allows re-rooting to an environment more than one step from the old root by essentially iterating the process described above. These functions are intended to work with a LISP interpreter like that found in the LISP 1.5 Programmer's Manual [1]. (Note, however, that boundary conditions require a null environment to bind NIL to NIL.) All variables are assumed to have *some* value in every environment; the more general case is left to the reader.

```
(DEFUN ASSOC (X A)
       (VCELL X))

(DEFUN APPLY (FN X A)
       (COND ((ATOM FN)
              (COND ((EQ FN 'CAR) (CAAR X))
                    ((EQ FN 'CDR) (CDAR X))
                    ((EQ FN 'CONS) (CONS (CAR X) (CADR X)))
                    ((EQ FN 'ATOM) (ATOM (CAR X)))
                    ((EQ FN 'EQ) (EQ (CAR X) (CADR X)))
                    (T (APPLY (EVAL FN A) X A))))
             ((EQ (CAR FN) 'LAMBDA)
              (PROG1 (EVAL (CADDR FN)
                           (RE-ROOT
                             (PAIRLIS (CADR FN) X A)))
                     (RE-ROOT A)))
             ((EQ (CAR FN) 'FUNARG)
              (PROG1 (APPLY (CADR FN) X (CADDR FN))
                     (RE-ROOT A)))))

(DEFUN RE-ROOT (A)
       (PROG (P Q)
             (SETQ Q (NREVERSE A))
             (SETQ P (CDR Q))
        LOOP (COND ((EQ Q A)
                    (RPLACA (CAR A) NIL)
                    (RPLACD (CAR A) NIL)
                    (RETURN A)))
             (RPLACA (CAR Q) (CAAR P))
             (RPLACD (CAR Q) (CDR (VCELL (CAAR P))))
             (RPLACD (VCELL (CAAR P)) (CDAR P))
             (SETQ Q P)
             (SETQ P (CDR P))
             (GO LOOP)))
```

(DEFUN f p b) defines a function f with parameters p and body b in the top-level environment.

(VCELL y) returns the value cell of variable y, which is a cons cell having the value as its CDR.

(NREVERSE x) reverses a list x destructively, by flipping the CDR pointers.

(PROG1 e1 e2 ...) evaluates e1, e2, ... in that order and returns the value of e1.

Figure 2. Continuous shallow binding interpreter

The re-rooting algorithm is quite simple. Suppose that e is the current root of the environment tree and e' is any other node in the tree which we wish to become the new root. Now, since e is the current root of the tree, the "parent" path from every other node in the tree will terminate at e, and we have a directed path from e' to e. We make two passes over that path, one forwards and one backwards. On the first pass, we go from e' to e while reversing all of the parent pointers. On the second pass, we go from e back to e' while exchanging the values in the association pairs with those in the value cells. This has the effect of shifting the values in the path over by one occurrence and terminating with the proper values for e' in the value cells.

This algorithm has much in common with the Deutch-Waite-Schorr marking algorithm for Lisp-style garbage collection [4] in that no additional storage is used, and pointers that previously pointed to a node's son are changed to point to that node's parent. Unlike that algorithm, the pointers are not changed back, unless of course the context is switched back to a previous environment.

The continuous shallow binding interpreter presented here is equivalent to the MACLISP interpreter for that class of LISP programs which do not have FUNARGs in the sense that the tree environment created will consist of a single filament which is isomorphic to the MACLISP *specpdl* (for *spec*ial variable *p*ush-*d*own *l*ist). However, our implementation of the model, being simple for pedagogic reasons, is not meant to be the most efficient way to perform shallow binding; storage reclamation, for example, is much less efficient than using a push-down stack.

Suppose now that we have a standard deep binding LISP 1.5 interpreter which has been augmented only by the `(SHALLOW)` primitive (see Fig. 3). When this primitive is invoked, the environment tree will be re-rooted to the current environment. Re-rooting transforms the environment tree in such a way that `(ASSOC X A)` is left invariant; therefore the interpretation proceeds correctly, but the length of the searches is changed. In other words, we can re-root at any point in the computation, not just during context-switching.

```
(DEFUN ASSOC (X A)
       (COND ((NULL A) (VCELL X))
             ((EQ (CAAR A) X) (CAR A))
             (T (ASSOC X (CDR A)))))

(DEFUN APPLY (FN X A)
       (COND ((ATOM FN)
              (COND ((EQ FN 'CAR) (CAAR X))
                    ((EQ FN 'CDR) (CDAR X))
                    ((EQ FN 'CONS) (CONS (CAR X) (CADR X)))
                    ((EQ FN 'ATOM) (ATOM (CAR X)))
                    ((EQ FN 'EQ) (EQ (CAR X) (CADR X)))
                    ((EQ FN 'SHALLOW) (PROG2 (RE-ROOT A) T))
                    (T (APPLY (EVAL FN A) X A))))
             ((EQ (CAR FN) 'LAMBDA)
              (EVAL (CADDR FN) (PAIRLIS (CADR FN) X A)))
             ((EQ (CAR FN) 'FUNARG)
              (APPLY (CADR FN) X (CADDR FN)))))
```

Figure 3. Occasional shallow binding interpreter.

We see that re-rooting is a transformation on the environment structure which preserves `(ASSOC X A)` and has nothing to do with *how* the structure was created. As a result, the interpreter could do either dynamic or lexical free variable binding and still be able to re-root. However, lexical binding interpreters may have nothing to gain from re-rooting, since there exist schemes for them such as Dijkstra's *display* [5] in which variable lookup time is bounded by a constant. A display cannot be used in LISP, though, because there it is impossible to tell *a priori* which variable occurrences are to refer to the same storage location.

Giving the programmer a choice as to which functions are to run shallow or deep bound can produce more efficient programs. For example, a tight loop may run faster when shallow bound, while an interrupt handler might run better deep bound, since re-rooting can be time consuming.

LISP 1.5 can be augmented with primitives for multiprocessing such as *fork*, P, and V. Several processes can be active in the environment tree in such a system without conflict because none of the processes changes the backbone of the tree. Since re-rooting preserves the value of `(ASSOC X A)`, executing `(SHALLOW)` in any process cannot affect the other processes so long as re-rooting is indivisible. Of course, if more than one process tries to do continuous shallow binding, then we will get a form of *thrashing* in which the processor spends all of its time re-rooting!

Our algorithm was discovered by pondering the Greenblatt Lisp machine proposal [6] which includes a shallow scheme for handling general funargs. His scheme does not reverse any pointers, but appends an up/down bit to each node in the environment tree which keeps the information as to which pointers would be reversed in our scheme. His scheme *must* continuously shallow bind because the pointers in the "down" path point in the wrong direction to be used by `ASSOC`. Our scheme both simplifies and generalizes his so that not only can shallow-binding be implemented more uniformly, but we also get the serendipidous benefit of being able to *choose* at any point between shallow and deep binding.

Our scheme can also be compared with the *cache scheme* developed by C. Hewitt, C. Reeve, and G. Sussman for the MDL language [7]. Their scheme associates, in addition to the value cell, a *cache cell* with each variable which is a pair consisting of a value and a pointer to an environment in which that value is valid. To access the value of a variable, the current environment pointer is compared with that in the cache cell and if they are equal, the cache value is used; otherwise, a search is made in the environment. In either case, the cache cell is updated to reflect its value in the current environment. A version of `ASSOC` that implements this approach is given in figure 4.

```
(DEFUN ASSOC (X A)
       (PROG (V)
             (SETQ V (ASSOC1 X A))
             (RPLACA (CCELL X) V)
             (RPLACD (CCELL X) A)
             (RETURN V)))

(DEFUN ASSOC1 (X A)
       (COND ((EQ A (CDR (CCELL X)))
              (CAR (CCELL X)))
             ((NULL A) (VCELL X))
             ((EQ (CAAR A) X) (CAR A))
             (T (ASSOC1 X (CDR A)))))
```

(CCELL y) returns the cache cell of variable y, which is a cons cell having a value cell as its CAR and an environment as its CDR.

Figure 4. Cache cell interpreter.

The cache scheme is independent of re-rooting in the sense that one, the other, or both can be implemented together. (Of course, with *continuous* shallow-binding, the cache is superfluous.) We claim that cache and occasional re-rooting are incomparable; i.e., complementary, in the sense that there exist programs that would run faster on a re-rooting scheme than on a cache scheme and vice versa. On a major context-switch involving many variables, a cache will perform poorly because only one variable will be updated at a time and searches will have to be made for each different variable before the interpreter is running truly shallowly. Re-rooting, on the other hand, will switch all of the deep variables to be shallow in one operation, thus economizing on searches. The cache will perform better in contexts where there are a large number of variables which *could be* accessed, but only a small fraction ever *are*. As our programs exhibit, both schemes are compatible, and together they can minimize variable access time.

References

1. McCarthy, J., Abrahams, P., Edwards, D., Hart, T., and Levin, M. *LISP 1.5 Programmer's Manual.* The M.I.T. Press, Cambridge, Mass., 1965, especially pp. 12-13, 70-71.

2. Moon, D. *MACLISP Reference Manual, Revision 0.* Project MAC, M.I.T., Cambridge, Mass., 1974.

3. Moses, J. *The Function of FUNCTION in LISP.* Memo 199, M.I.T. A.I. Lab., Cambridge, Mass., June, 1970.

4. Knuth, D. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968, p. 417.

5. Randell, B. and Russell, L. J. *ALGOL 60 Implementation.* Academic Press, London and New York, 1964, pp. 65-68.

6. Greenblatt, R. The Lisp Machine. A.I. Working Paper 79, M.I.T. A.I. Lab., Cambridge, Mass., November, 1974.

7. Galley, S. and Pfister, G. The MDL Language. Programming Technology Division SYS.11.01, Project MAC, M.I.T., Cambridge, Mass., September, 1975.