Computer Languages, Vol. 4, pp. 187 to 198
Pergamon Press Ltd 1979. Printed in Great Britain

DEEP AND SHALLOW BINDING: THE ASSIGNMENT OPERATION

HANAN SAMET

Computer Science Department, University of Maryland, College Park, MD 20742, U.S.A.

(*Received* 25 *September* 1978; *in revised form* 13 *February* 1979; *received for publication* 12 *April* 1979)

Abstract—Programming languages which use dynamic identification for free variables (i.e., non-local references) are generally implemented with a deep or shallow binding variable access strategy. In this paper, variable access through the binding environment is assumed to be desirable. Given this assumption, it is demonstrated that the act of assigning values to variables may yield unexpected results for some of the binding strategies when functional arguments and results are used. A number of variations of deep and shallow binding strategies are examined along with the modifications necessary to implement the assignment operation in the expected manner.

Deep binding Shallow binding Dynamic binding Access environments Functional arguments Binding environments Activation environments

1. INTRODUCTION

CONCURRENT with the growth of the field of artificial intelligence has been the development of special purpose languages [3, 14]. Many of these languages do not use simple lexical identification for determining values of non-local (i.e., free) variable references as in ALGOL60 [11]. Instead, dynamic identification is used as in LISP [9] or SNOBOL4 [7]. Therefore, variable access mechanisms cannot make use of the display [5] of ALGOL60. This has sparked an interest in conflicting binding strategies. In particular, attention is focussed on "deep binding" and "shallow binding" [1, 2]. Here, these strategies are briefly discussed in the context of the pitfalls associated with them when an assignment operation is performed. A comparison is also made of the various shallow binding implementations.

At this point it is appropriate to define some terms more closely. The term "environment" is used to denote a mapping from variable names to values rather than from variable names to locations. All of the examples assume a "call by value" [12] parameter transmittal mechanism with the exception of functional arguments and results (i.e., procedure names). Dynamic identification or dynamic binding stipulates that the most recent binding in the calling chain is used. In contrast, lexical identification or static binding stipulates that the static structure of the program at compile-time is used to determine the non-local environment (e.g. [11]). For example, consider the program fragment in Fig. 1. Of particular interest is the value of x which will be printed when the main program has called procedure P which in turn has called procedure Q. Using a static binding method, x has a value of 2 whereas a dynamic binding method yields a value of 1 for x. In general, dynamic binding methods are interesting because they correspond closely to intuitive ideas of computation and thus they are simple to understand.

2. ACTIVATION VS BINDING ENVIRONMENT

Figure 2 contains a set of functions expressed in an ALGOL-like notation for a language that uses dynamic identification for non-local variables. The example, similar to the one given in [16], is used in the subsequent discussion to illustrate the differences between the various binding strategies. In this example, $f4$ returns the function f as its result and it will eventually be invoked in $f6$. Suppose f is defined as follows:

```
procedure f;
return (if z = 8 then 1
        else -1);
```

```
begin
  local x;
  procedure Q;
  begin
    print(x);
      ...
  end;
  procedure P;
  begin
    local x;
    x: = 1;
    Q;
      ...
  end;
  x: = 2;
  P;
    ...
end;
```

Fig. 1. Sample program illustrating dynamic and static binding.

A key question facing language implementors is to which instance of the variable z does f refer? More generally, what bindings are associated with instances of free variables in functional arguments and results. Depending on the application, there are two basic choices.

In applications that make use of process communication (e.g. the "blackboard" in the Hearsay II system [8]), the activation environment is desirable since it enables making use of up-to-date information. In such a case, $z = 11$ is desired. In many LISP systems (e.g. [13]) this is achieved by use of the function QUOTE as shown in the revised definition of $f4$ below:

```
procedure f4(v,y,z);
begin
    ...
  return (QUOTE(f));
    ...
end;
```

In applications such as problem solving where remembering a context is important, the binding environment is preferable. In such a case, $z = 8$ is the desired value. This is implemented in many LISP systems (e.g. [13]) by use of the function FUNCTION as shown in the revised definition of $f4$ below:

```
procedure f4(v,y,z);
begin
    ...
  return(FUNCTION(f));
    ...
end;
```

Making use of the binding environment can be costly since the most recent binding of the variable is not always the one that is desired. This means that active environments must be stored and a more complicated search performed. In order to increase the efficiency of such a binding method, a concept is proposed in [15] which is used in the LISP machine [6] and termed "closure". In essence, the programmer must specify with each functional result (or argument), say f, the variables whose values are to be deter-

```
procedure f1(x);
begin
    . . .
  f2(2,3);
    . . .
end;
procedure f2(v,x);
begin
    . . .
  f3(4,5);
    . . .
end;
procedure f3(y, w);
begin
  local t;
    . . .
  t: = f4(6,7,8);
  f5(9,10,t);
    . . .
end;
procedure f4(v,y,z);
begin
    . . .
  return (f); /*binding environment*/
    . . .
end;
procedure f5(y,z,fn);
begin
    . . .
  f6(11,12,fn);
    . . .
end;
procedure f6(z,w,fm);
begin
    . . .
  fm; /*activation environment*/
    . . .
end;
```

Fig. 2. Sample program illustrating activation and binding environments.

mined by the binding environment. All instantiations of other variables within the particular instance of f default to the activation environment. For example, if the binding environment is to be associated with z in $f4$, then $f4$ would have the following definition:

```
procedure f4(v,y,z);
begin
    . . .
  return (CLOSURE((z),f));
    . . .
end;
```

The trouble with such a solution is that it is overly restrictive on the programmer. Frequently, there is no knowledge of which functions will be subsequently invoked. In such a case, values of all of the variables must be saved (this is equivalent to the FUNCTION construct). This is expensive especially in light of the fact that often most of

the variables will never be instantiated. For example, suppose f has the following definition:

```
procedure f;
begin
    ...
  if v = 5 then f1(1)
  else if v = 6 then f2(2,3)
  else f3(4,5);
    ...
end;
```

The remainder of the discussion assumes that the binding environment is the desired one. The following definition of f is used:

```
procedure f;
return (if x = 0 then 1
        else -1);
```

3. DEEP BINDING

Deep binding is implemented by the classical association list (also known as an a-list) approach of LISP. Briefly, a list of variable bindings is maintained in the form of name-value pairs. The list generally grows in a stack-like manner in that as soon as a function is entered, the bindings of its arguments are placed at the front of the list. Whenever the value of a variable is to be accessed, the list is searched for the most recent occurrence of the variable and the corresponding value is returned. Nevertheless, the stack is an inappropriate data structure. There are two problems [10]:

1. When a function is passed to another function as an argument, an environment is also transmitted. In the case of a stack, this is a pointer to a location on the stack where the search for variables is to commence. If storage for subsequent function calls is allocated starting at this location, then valid data of the caller's environment will be destroyed.
2. When a function is returned as the result of another function, an environment pointer must also be returned. This environment, at the time of the return, is on top of the stack. However, subsequent function calls may overwrite this area.

In summary, the problem is that the association list, although growing in a stack-like manner, must actually be a tree.* Nevertheless, even with tree-like implementations, there remain problems with respect to the effect of an assignment operation. In particular, in some instances when an assignment operation is performed, there is a question as to whether the new value associated with the variable should be subsequently used by the saved environment; or whether the previous value should be used with saved environments. The remainder of the paper focusses on this problem while analyzing various binding strategies and implementations.

For example, consider the set of functions in Fig. 2 and assume that $f1(1)$ has been called initially. Assume further that x has been set to 0 just prior to the invocation of fm in $f6$. Invocation of fm in $f6$ is equivalent to invocation of f. If an activation environment binding method is used, then there is no question that f accesses x with a value of 0 and thereby returns 1 as its value. However, when a binding environment binding method is used, there remains a question as to whether the correct binding of x is 3 or 0. We believe that consistency demands that use of a binding environment result in x having a value of 3 when accessed by f since this is the value of x in the environment in which the functional value was returned. Unfortunately, as will be seen, this is not always the case, and in many implementations a value for f of 1 will be returned rather than -1. The reason this problem arises here and not in the definition of f in Section 2, which accessed

*But see [4] where the tree is implemented by use of a collection of stacks.

variable z, is that here the assignment is made to an instance of x shared by both environments, whereas an assignment to z at the same location (i.e., just prior to the invocation of *fm* in $f6$) affects a different instance of z than the one accessed by f (i.e., the assignment affects z of $f6$ whereas f refers to z of $f4$).

In an implementation of deep binding which makes use of an association list, a destructive assignment operation results in overwriting the old binding. The problem arises, in part, from a desire to save storage by use of the following LISP-like implementation of an assignment function. Assume that ALISTG corresponds to the current environment. ASSIGN1 results in the assignment of the value of B to variable A. Note that ALIST corresponds to the current environment—i.e., it is bound to ALISTG.

```
procedure ASSIGN1(A,B,ALIST);
begin
  if null(ALIST) then error
  else if first (first(ALIST)) eq A then
    rest (first(ALIST)): = B /* same as rplacd (car(ALIST),B)*/
  else ASSIGN1(A,B,rest(ALIST));
end;
```

In order to obtain the desired result, whenever an assignment is made to any variable, a copy must be made of the association list starting at the modified variable. ASSIGN2 achieves this result. It is invoked in the same manner as ASSIGN1. Once again, ALISTG corresponds to the current environment. A difference between ASSIGN1 and ASSIGN2 is that the value of ASSIGN2 is the new value of the current environment which must be assigned to ALISTG—i.e., the invoking call is of the form "ALISTG: = ASSIGN2 (*A*,*B*,ALISTG);."

```
procedure ASSIGN2(A,B,ALIST);
begin
  if null(ALIST) then error
  else if first(first(ALIST)) eq A then
    addtolist(makepair(A,B), ALIST)
  else addtolist(first(ALIST),ASSIGN2(A,B,rest(ALIST)));
end;
```

The above solution works because the LISP storage allocation mechanism obtains a new LISP cell each time a CONS operation is performed (e.g., makepair and addtolist in our example). In particular, in the step "addtolist(first(ALIST),ASSIGN2(*A*,*B*,rest (ALIST)))", the cell corresponding to first(ALIST), which denotes a name-value pair, is reused; however, a new cell is obtained by addtolist. This is in contrast with the destructive ASSIGN 1 operation where the name-value pair was modified but the list links remained the same.

4. SHALLOW BINDING

Shallow binding is an attempt to avoid the costly search associated with the variable lookup process in deep binding systems. Using such an approach, the value of a variable may be obtained in one memory fetch. This relies on a specified location, the value cell, in which the current binding of the variable can always be found. The cost of such an approach is that, unlike deep binding systems, whenever a function is invoked, the value cells of the argument names and locally declared variables must be saved. Similarly, upon function exit, they must be restored. Moreover, in the case of a functional argument, an entire environment (i.e., the contents of all of the value cells) must be saved and restored.

There are several implementations for shallow binding. In the remaining sections, a number of them are examined with the aid of the example in Fig. 2. Again, assume that $f1(1)$ has been called initially. Let $E1$ denote the environment associated with f when it is returned as the value of $f4$. Let $E2$ denote the environment associated with $f6$ when *fm* is invoked (*fm* is actually bound to f). Figure 3 is a tree-like representation [16] of the state of the computation at $E2$. The term frame is used to denote the collection of formal and

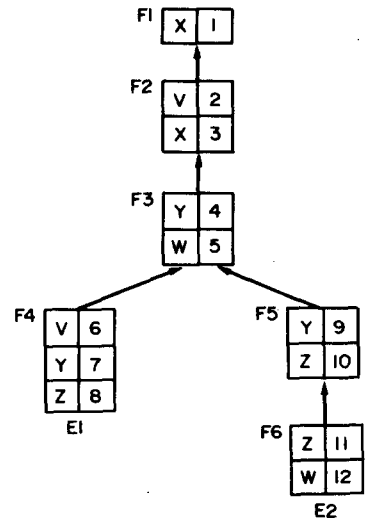


Fig. 3. Tree-like representation of Fig. 2.

local variables associated with each function activation comprising an environment. $F1, F2, F3, F4, F5$, and $F6$ are the frames in Fig. 3. The set of frames associated with a particular environment is said to be a frame chain. $F1, F2, F3, F4$ and $F1, F2, F3, F5, F6$ are the frame chains associated with environments E1 and E2 respectively in Fig. 3.

4.1 *Indirect value*

This method [16] associates with each variable a value cell whose contents is a pointer to the variable's binding in the current environment—i.e., a pointer to the variable's location in the frame in which its current binding is found. Figure 4 shows the application of this method to Fig. 3 when environment $E2$ is active.

Environment switching requires updating the value cells and is accomplished by traversing the frame chain of the new environment until all entries in the value cells have been updated (e.g. $F4$, $F3$, and $F2$ when switching from $E2$ to $E1$).

One improvement to the above environment switching strategy is a variation on a scheme of [4] which updates a value cell only when a reference occurs. In such a case, when searching for a variable's binding, update the value cells of all variables encoun-

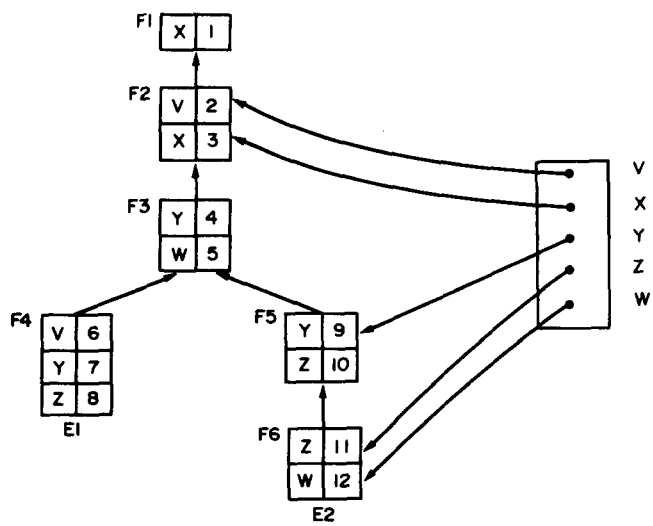


Fig. 4. Environment $E2$ using the indirect value method.

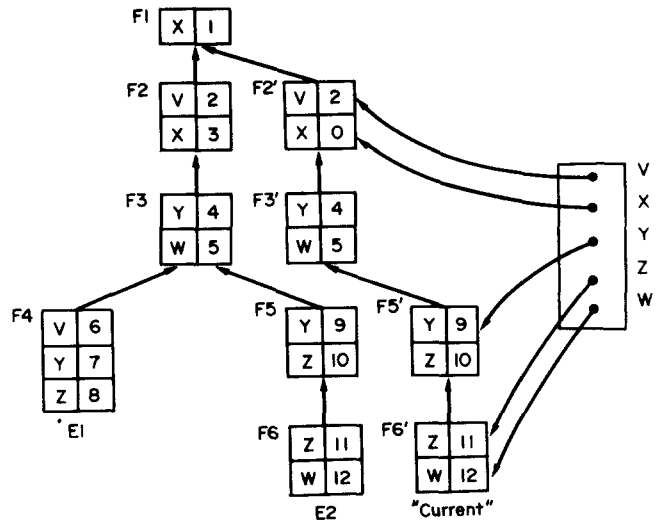


Fig. 5. Figure 4 after the assignment of 0 to x.

tered during the search for which the value cells do not contain the appropriate values. The next time a reference to an improperly bound variable occurs in the same frame, the search for its current binding resumes at the frame in which it was terminated during the previous search. Otherwise, the search must be restarted in the current frame. For still another more efficient implementation of the indirect value method see [16] where all frames in which a variable is bound are linked.

Unfortunately, assignment statements are destructive in the sense that a variable's frame entry in the current environment will be overwritten. Thus the method is plagued by the same problem associated with the classical deep binding implementation examined earlier. For example, the assignment x: = 0 in $f6$ just prior to invocation of fm in $f6$ will result in f returning a value of 1 instead of the desired -1. The problem can be alleviated by using the same technique that was employed in the deep binding scheme. i.e., a copy is made of all frames invoked in the current environment subsequent to, and including, the frame associated with the variable whose value is being changed (e.g. $F2, F3, F5$ and $F6$). In addition, the links in the value cells pointing at frames $F2$, $F3, F5$, and $F6$ must be changed to point at frames $F2', F3', F5'$, and $F6'$. Figure 5 shows how Fig. 4 would be modifed to reflect the assignment of 0 to x. Note that if copies of frames $F3, F5$, and $F6$ were not made (i.e., because none of their components changed in value), then a problem may arise should frame $F5$ be the current frame in a previously saved environment. In this case, $F5$ will be invoked with x incorrectly being 0 (because $F5$ is linked to $F3$ which is linked to $F2'$ rather than to $F2$) whereas the environment was saved with x being 3.

4.2 *Direct value*

In this method [13, 16], the value cell always contains the current value of the variable. All frames along the current frame chain contain the name of the variable that is bound by the frame and its immediately previous value. Whenever a variable is bound (i.e., a new frame is entered), the name of the variable and its current value (i.e., the value in the value cell) are placed in the frame and the new value is placed in the value cell. When a frame is exited, the appropriate value cells are set to the contents of the frame. Figure 6 shows the application of this method to Fig. 2 when environment $E2$ is active.

Environment switching requires saving the value cells in the appropriate locations and restoring the desired environment. This is accomplished by "unwinding" (see the definition below) the frame chain of the current environment until a frame (termed the "intersecting frame") is reached which is common to both environments (e.g. frame $F3$ when switching from $E2$ to $E1$ and unwinding the chain consisting of $F6, F5$, and $F3$). For all

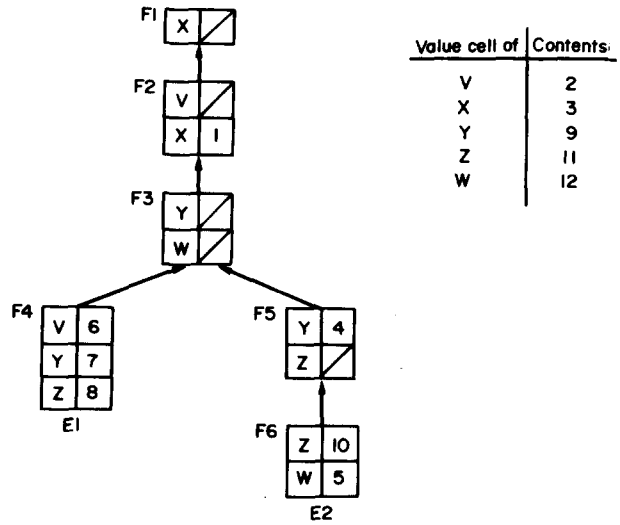


Fig. 6. Environment $E2$ using the direct value method.

frames other than the intersecting frame, unwinding means the exchange of the contents of the value cells with the variable bindings in the frames. Once the intersecting frame is reached (its identity can be determined from the frame chains associated with each environment), the process is reversed in the sense that the frame chain of the switched-to-environment is traversed, starting with the successor of the intersecting frame, and in the process the contents of the value cells are exchanged with the variable bindings in the frames (e.g. starting at frame $F3$, traverse the frame chain consisting of $F4$ when switching from $E2$ to $E1$). Figure 7 shows the result of switching from $E2$ to $E1$. This technique is attributed to R. Greenblatt by [16].

Assignment statements pose the same problem as in the indirect value method. The only appropriate solution is to make a copy of the frames in the current frame chain starting with the current frame up to and including the frame in which the variable was most recently bound. This process is analogous to the unwinding step of environment switching, where the analogue of the "intersecting frame" is the frame prior to the most recent frame in which the variable was bound. However, as a frame is copied, it is unwound. Once the frame in which the variable was most recently bound is reached (e.g. $F2$ in Fig. 6), the direction of the unwinding process is reversed and the new chain is

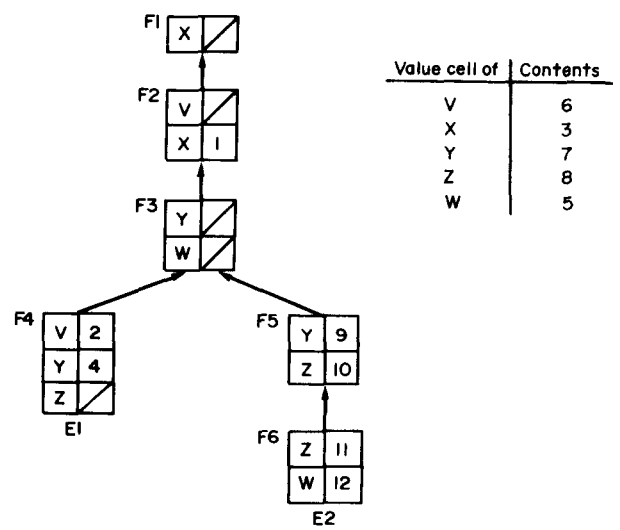


Fig. 7. Result of switching from environment $E2$ to $E1$ using the direct value method.

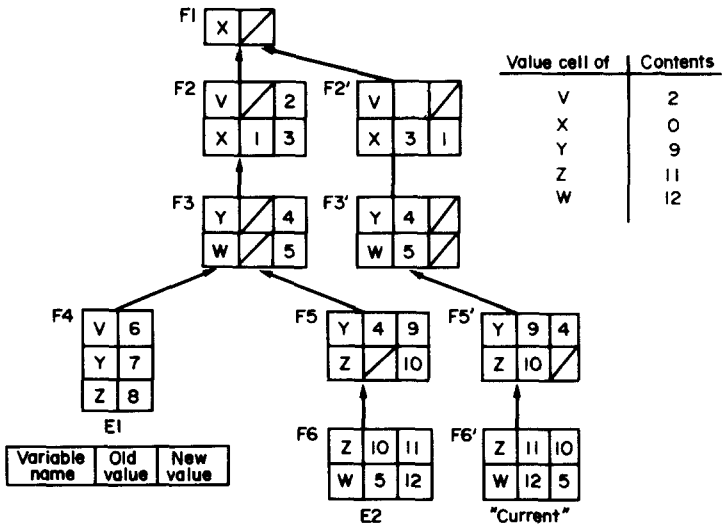


Fig. 8. Figure 6 after the assignment of 0 to x.

traversed starting with that frame (e.g. $F2$ in Fig. 6). Figure 8 shows how Fig. 6 is modified to reflect the assignment of 0 to x just prior to the invocation of *fm* in *f*6 (e.g. in environment $E2$). For each frame that is being traversed, its initial contents and its new contents (labeled old and new) are shown—i.e., frames $F6$, $F5$, $F3$, $F2$, $F2'$, $F3'$, $F5'$, and $F6'$ have been traversed in the designated order.

Two additional items are worthy of note. First, in the unwinding process the value cells will be changed. However, once the environment has been unwound (e.g. $E2$), the value cells will be restored to their contents prior to the unwinding step (with the exception of the variable whose value was changed by the assignment statement) by virtue of the traversal of the new frame chain. Second, when switching environments, the unwinding process leaves the "intersecting" frame unchanged because its values refer to instances of variables occurring in previous frames. Note that in the case of an assignment statement, the frame corresponding to the most recent binding of the variable is modified because it must be set to the value which the assignment statement has just overwritten (e.g., 3 in frame $F2$ in Fig. 8) since this environment is no longer the current environment (e.g., environment $E2$ in Fig. 8).

4.3 *Frame table method*

In [17] a method is presented which makes use of frame tables i.e., one table for each variable. Entries in the frame table are frame-value pairs i.e., one pair for each frame in which the variable is bound. Figure 9 shows the application of this method to Fig. 2 just prior to the invocation of *fm* in *f*6 (e.g. in environment $E2$). In addition to the frame table there exist value cells for the various variables. Each value cell contains a frame tag indicating the frame with which the current value cell entry is associated. Accessing a variable may be a costly process if its current binding is not in the value cell. In such a case, the frame table associated with the variable is searched for a binding with the current frame. If none is found, then the table is searched again for a binding with the dynamically preceding frame, etc. (it is assumed that the table is not sorted). Switching

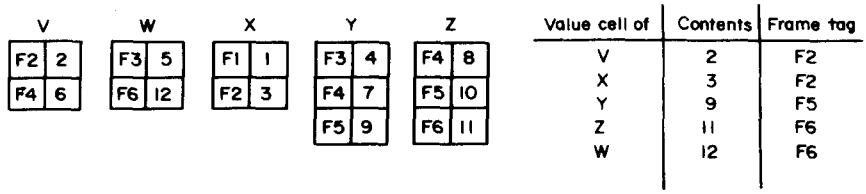
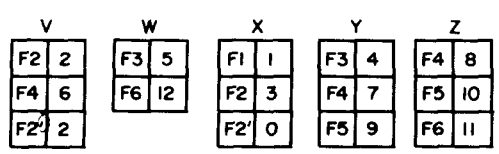


Fig. 9. Environment $E2$ using the frame table method.



Value cell of	Contents	Frame tag
V	2	F2'
X	0	F2'
Y	9	F5
Z	11	F6
W	12	F6

Fig. 10. Figure 9 after the assignment of 0 to x.

environments, as in deep binding methods, requires no extra work other than use of a different frame chain.

Unlike other binding implementations, the assignment statement poses no problem when using the frame table method. A new frame is created to replace the frame in which the variable was previously bound and a new value pair entry is added to the variable's frame table. The same process must be performed for the remaining variables bound in the frame of the variable whose value has changed. Figure 10 shows how Fig. 9 would be modified to reflect the assignment of 0 to x just prior to the invocation of *fm* in *f*6 (e.g. in environment $E2$). Note that a new frame, $F2'$, has been created and the frame tables of x and v have been modified accordingly. Of course, the current frame chain must be modified to contain $F2'$ instead of $F2$ i.e., the current frame chain becomes $F1, F2', F3, F5$, and $F6$ instead of $F1, F2, F3, F5$, and $F6$. Also observe that the assignment statement causes the frame chain to become unsorted (i.e., the new frame will generally have a higher internal number associated with it for identification purposes than all previously generated frames) thereby requiring a full search through the frame table when attempting to access variables.

4.4 *Comparison*

At this point it is appropriate to compare the various binding strategies. The relative merits of deep and shallow binding have been discussed earlier. With respect to the shallow binding methods, the following observations can be made.

For variable access, the direct and indirect value methods are superior to the frame table method since the value cell always contains the current binding. When using the frame table method, the value cells do not always contain the current binding of all variables. In such a case, the frame table associated with the specific variable must be searched. If there are many instances of a variable (i.e., it has been rebound or reassigned many times), then the table may be large and unless it is sorted, the search may be time consuming.

The indirect value and frame table methods require more work to restore the value cells when a frame is exited than does the direct value method. Recall, in the direct value method such information is immediately accessible while the other methods require a search. However, this deficiency can be alleviated in both the indirect value and frame table methods by only restoring value cells when their values are actually needed [4]. Alternatively, the indirect value method can be improved by adding a link field to each variable's frame entry pointing at its previous binding [16].

Environment switching is achieved most efficiently by the frame table method. However, a high price must be paid for subsequent variable accesses. When environment switching and general function exit are taken into account, the indirect value method is more efficient than the direct value method. There are a number of reasons. First, the value cells need not be immediately updated to their new values when the indirect value method is used. Instead, the value cells need only be updated when their values are needed. Similarly, link fields may be used to speed up the search. Second, when the direct value method is used, the current environment must be unwound to an intersecting frame followed by a reversal of the process in the new environment. This is in contrast with the indirect value method where only one unwinding step needs to be performed. Nevertheless, there are cases where the direct value method is more efficient than the indirect

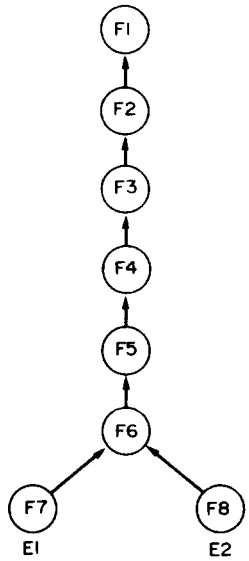


Fig. 11. Sample environments.

value method. In particular, if there aren't too many frames to unwind, as in Fig. 11 ($F1$–$F8$ are frames), then a switch from environment $E2$ to $E1$ might be faster when using the direct value method. Finally, note that updating value cells only when a binding is needed is not very practical for switching environments when using the direct value method since the unwinding and winding process can not be avoided—i.e., one must still traverse all of the frames in which the variable is bound. This is true even if successive frames containing bindings for the desired variable are linked (in such a case, the list of bindings for the variable must be traversed).

With respect to the assignment statement, the frame table method is the most efficient. In this case, only the frame containing the variable whose binding is being modified needs to be copied (equivalently, the frame tables of all variables bound in this frame must have an additional frame-value pair). The indirect value method is more efficient than the direct value method since, although in both cases the frames between the current frame and the frame in which the newly assigned to variable was previously bound need to be copied, the direct value method requires retraversing the frames between the current frame and the frame where the variable was most recently bound. The key to the frame table method is that the frame chain is separated from the data. Thus there is no need to copy the frames as is the case when the indirect and direct value methods are used. Note that if the indirect value method is implemented with a frame chain instead of direct links, the need to copy frames can be avoided and then the indirect value method becomes very similar to the frame table method.

5. CONCLUSION

In summary, it appears that the frame table method is the most efficient shallow binding implementation method provided variable accesses are infrequent—i.e., whenever environment changes occur, not all variables are accessed. Introduction of the assignment statement and its subsequent solution demonstrated the superiority of the indirect value method to the direct value method. Given little information about a program to be executed, the indirect value method with modifications such as an extra link field linking frames in which a variable is bound is probably the most efficient since the presence of the frame chain with the data avoids costly searches.

Acknowledgements—Special thanks go to John R. Allen who patiently served as an initial sounding board, and also to Phil Agre and Robert Kirby for their valuable comments.

REFERENCES

1. J. R. Allen, *Anatomy of LISP* p. 152. McGraw–Hill, New York (1978).
2. H. G. Baker Jr., Shallow binding in LISP 1.5, *Comm. ACM* **21**, 565–569 (1978).
3. D. G. Bobrow and B. Raphael, New programming languages for artificial intelligence, *ACM Computing Surveys* **6**, 153–174 (1974).
4. D. G. Bobrow and B. Wegbreit, A model and stack implementation of multiple environments, *Comm. ACM* **16**, 591–603 (1973).
5. E. W. Dijkstra, Recursive programming, *Num. Math* **2**, 312–318 (1960). Also in S. Rosen (ed.), *Programming systems and languages*. McGraw–Hill, New York (1967).
6. R. Greenblatt, The LISP machine, Massachusetts Institute of Technology Internal Documentation (1977).
7. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language* (Second Edition), Prentice-Hall, Englewood Cliffs, New Jersey (1971).
8. V. Lesser and L. Erman, A. retrospective view of the Hearsay-II architecture, *Proceedings of the Fifth International Joint Conference on Artifical Intelligence*, Boston, MA, 790–800 (1977).
9. J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM* **3**, 184–195 (1960).
10. J. Moses, The function of FUNCTION in LISP, *SIGSAM Bull.* **15**, 13–27 (1970).
11. P. Naur (Ed.), Revised report on the algorithmic language ALGOL 60, *Comm. ACM* **3**, 299–314 (1960).
12. T. W. Pratt, *Programming Languages: Design and Implementation*. Prentice-Hall, Englewood Cliffs, N.J. (1975).
13. L. H. Quam and W. Diffie, Stanford LISP 1.6 manual, Stanford Artificial Intelligence Project Operating Note 28.7, Computer Science Department, Stanford University (1972).
14. C. J. Rieger, J. Rosenberg and H. Samet, Artificial intelligence programming languages for computer aided manufacturing, *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC-9**, 205–226 (1979).
15. E. Sandewall, A proposed solution to the FUNARG problem, *SIGSAM Bull.* **17**, 29–42 (1971).
16. J. Urmi, A shallow binding scheme for fast environment changing in a "Spaghetti Stack" LISP system, Report Li TH–MAT–R–76–18, Linkoping University, Sweden (1976).
17. B. Wegbreit, Retrieval from context trees, *Inf. Processing Lett.* **3**, 119–120 (1975).

About the Author—HANAN SAMET received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

Since 1975 he has been an Assistant Professor of Computer Science at the University of Maryland, College Park. His research interests are data structures, programming languages, code optimization, data base management systems, and artificial intelligence.