An Alternative Dynamic Binding Model: Deep Binding With Cacheing

Cyril N. Alberga
T. J. Watson Research Center
IBM
alberga@ibm.com

The two most common dynamic binding models used in Lisp systems are shallow and deep binding. For a succinct summery of these see Padget and Fitch [1985], which includes a proposed third model. It is often claimed that shallow binding are unsuited to systems which allow context switching. We wish to describe an alternative binding model, deep binding with cacheing, which provides the efficiency of shallow binding in "ordinary" call/return control, while exhibiting the ease of context switching of deep binding. This binding model was implemented in the Lisp system built at the IBM Research Center, later elaborated into LISP/VM. In this article the term LISP/VM will be used to encompass both that program and all of its progenitors developed at IBM Research.

This model is not original to LISP/VM. The original proposal is included in Bobrow and Wegbreit [1973], which was the inspiration for the LISP/VM stack model. Despite this early presentation the model seems to have been forgotten. Even White [1978] in his description of LISP/370, an early version of LISP/VM, ignores the validation problem. While in White [1982] Other descriptions of shallow binding systems, [Teitelman, 1978, Bates et al., 1982], clearly are refering to ordinary shallow bindings, and these in the context of implementations of INTERLISP. In light of this neglect I wish to present the technique in some detail.

The Stack Model

The problems of dynamic binding discussed in this article are relevant in at least two stack models. These can be thought of as the SCHEME stack and the LISP/VM-INTERLISP stack. Both models support a stack which has the structure of an inverted tree, that is each node points to its ancestor, with all pointers converging on a single root node. At all times there is one leaf of this tree which is the active stack frame. All other leaves are retained frames, captured by some data object. In SCHEME this is a continuation, in INTERLISP an POS, in LISP/VM a state descriptor (SD). All of these models assume a two part stack frame, one part containing the variable bindings (value cells, not values) which are shared by all descendant frames, one part containing the control information, unique to each branch. INTERLISP uses the terms basic frame and frame extension, LISP/VM uses head-of-environment and dump. It is the dump portion of the stack frame which is pointed to by the SD, and which forms the leaves of the tree.

In the SCHEME model, where closures are only able to capture the immediate lexical environment, dynamic bindings are only effected by shifts of control between leaves of the stack. In this model there is a single set of links between stack frames.

In the INTERLISP and LISP/VM models, with their more complete environments, there are always a pair of links between frames, the access link (A-link) and the control link (C-link). Each system has mechanisms which result in frames with different access and control ancestors. Thus there are two trees imposed on the collection of stack frames.

Other binding models

What other binding models can be used when context switching is allowed? Note that it is only the binding of dynamic variables which are under consideration here. In systems, such as LISP/VM and INTERLISP, which construct full closures, capturing the entire environment, lexical and dynamic, evaluation or application with respect to an environment (funarg application) are effected. In systems, such as SCHEME and Common Lisp, which only support lexical closures, these operations are not effected. Finally, for systems such as LISP/VM, INTERLISP and SCHEME (but not for Common Lisp) which support continuations, application of continuations is also effected.

How can dynamic bindings be supported in the systems in question? Pure deep binding is correct in all instances. This is tautological, as variable evaluation in Lisp is defined in terms of a deep binding model, with all other binding models simply attempts to mimic deep binding with more efficient value access. The most generally used alternative is shallow binding with fixup.

In this model dynamic binding involves the saving of the current contents of the shallow binding cell of a variable, and the updating of that cell with the new value. Upon exiting a stack frame which has bound a dynamic variable the old value is restored. When the context is switched, by the application of a continuation or by evaluation or application with respect to a complete environment, the least common ancestor of the current stack frame and the stack frame representing the new environment is found. The stack is then traced from the current frame to the LCA, with all dynamic variables restored, as if the frames were being exited. Then the new branch of the stack is traversed, from the LCA to the frame which is to be resumed or used for evaluation. During this traversal all dynamic variables are rebound. A discussion of variants of the shallow binding with fixup model, with implementations in terms of continuations will be found in Clinger [1987].

If this operation need only be done when control is transferred (as in SCHEME), and if the time spent in any one locus of control is relatively long, then this model may be quite satisfactory. If, however, the fixup must be done on any evaluation with respect to an environment (such as a simple call to EVAL with a variable and an SD), then it is probably excessive. In many situations the overhead of the fixup will exceed the cost of deep binding searches.

A second model is that presented by Padget and Fitch [1985]. From his studies this is probably a viable alternative, giving almost the efficiency of shallow binding in the simple LIFO control case, without any extra overhead during context switching. It does, however, involve a somewhat complex labeling scheme to be applied to at least every frame which is retained or which binds a dynamic variable.

Deep Binding With Cacheing

The LISP/VM stack

LISP/VM will be used to encompass both that program and all of its progenitors developed at IBM Research, see [Alberga et al., 1986, Blair, 1976., IBM Corporation, 1978, IBM Corporation, 1984]. The only characteristic of the system of interest in this article is its stack structure and stack handling operations.

LISP/VM supports a framed stack with full retention. Unlike some later Lisp, such as Common Lisp, closures capture the entire environment, including dynamic variables. A primitive data object, the state descriptor (SD), serves to record both the environment and the control. An SD may occur as one component of a closure, may be given as an optional argument to EVAL and APPLY, or my be applied to an argument to provide the facilities of the continuations of SCHEME. The process of applying a SD, causing control to move laterally in the tree of stack frames, is referred to as "jaunting" (Bester). (See Appendix I for the relationship between the state descriptor creating function, STATE, and the continuation creating function CALL/CC.)

There is an important difference between the captured environment and the captured control. The environment which is captured is a set of bindings, not a set of values. That is, if a fluid binding for some variable, say X, occurs in a stack frame which is ancestral to two or more environments, than an assignment to (the free variable) X in any of those environments effects all of them. The control component of a stack frame, however, is not shared. If execution is resumed in a saved state control passes to the exact expression in which the state descriptor was originally created, albeit with possibly modified values of variables.

This distinction between environment and control is maintained by splitting the stack frame into two components, the head of environment (head-of-E), and the dump (control). The head-of-E contains the value cells, while the dump contains the instruction counter, the working stack, and certain other sub-components.

In a program with a strict FIFO calling discipline the head-of-E and dump are contiguous, and may be discarded on exit from a frame. In the presence of state saving this is not true. To provide for this contingency each dump contains a pointer to its associated head-of-E. In order that the control information be preserved the dump portion of a saved state is never actually used for execution. Rather, upon the creation of a state descriptor pointing to a particular frame, the dump of that frame is copied to the stack frontier and used for subsequent execution.

The dump portion of a stack frame will be copied in two other situations. After a state descriptor has been constructed all of the ancestor frames of the current frame may no longer be used as is. Upon attempting to return to such a frame it will be found that its dump is no longer at the stack frontier, but must be copied to be used. (The garbage collector attempts to rearrange the stack to place the currently active chain of stack frames at the frontier of the compacted stack.) The second case involves the resumption of control at a saved state. Even in the unlikely case that the dump for the captured frame were at the frontier it still must be copied, in order to preserve the control information to allow additional resumptions.

The evaluation of expressions with respect to an SD and the application of funargs does not require the copying of any frames. It does, however, result in a structure for the environment which differs from the control. Both are inverted trees, that is trees with the links running from the leaves to the roots. However, the nodes which correspond to an evaluation with respect to an SD have different ancestors for their environment and control chains.

One of the goals of the IBM Research Lisp development group was to eliminate some of the inconsistences which had crept into Lisp over the years. To that end the treatment of lexical and fluid ("special") variables was unified. Both types of variables are bound to value cells in stack frames. Each stack frames has an associated display which indicates which bindings are lexical and which are fluid. With no other additions to the binding and evaluation machinery LISP/VM would be a typical deep binding system. The expense of variable evaluation implied (somewhat greater than normal, as the search must pass through all stack frames, rather than just an a-list of dynamic bindings) was felt to be unacceptable.

It was clear from the start that ordinary shallow binding would not work in the presence of context switching, whether done by jaunting or by the evaluation of expressions with respect to a saved environment. Following a suggestion in Bobrow and Wegbreit [1973] it was decided to associate a control block (erroneously called a shallow binding cell, SBC, in the LISP/VM descriptions) with each identifier (symbol). This control block contains a pointer to a value cell, on the stack or, for non-stack bound (global) variables, in an a-list, and a marker, the activation chain descriptor (ACD) which is used to test, conservatively, for the currency of the binding.

The model of binding used in LISP/VM has certain similarities with that of Padget. As in Padget's binding model labels are given to frames in the stack, and these labels are used to identify the currently valid binding of a free variable. Unlike Padget's model labels need only be given to a subset of the extant frames. All fluid variable bound in a frame are associated with the ACD of one such ancestor dump.

The major differences between the LISP/VM model and that of Padget are the lack of ordering of the labels, and their use in verifying the currency of a binding cache cell, rather than in identifying the binding itself.

In the simplest form of the model the ACD of the nearest labeled frame is used as the current ACD. When a variable is to be evaluated its cache cell is examined and if the stored ACD matches the current ACD the binding indicated by the cache cell is used. Otherwise the stack is search, the cache cell is refreshed, and the located binding used. When a fluid variable is bound the associated cache cell must either be reset to point to the new binding, or spoiled, by setting the ACD slot to an invalid value. In the former case the old contents of the cache cell (the pointer to the binding, not its value) may be saved, to be restored upon unbinding, or the cache cell maybe spoiled on exit.

Spoiling will trigger a deep binding style search on first reference, after which the cache cell will be valid again.

Every time control passes through a labeled frame, or reverts to a different branch of the stack, the current ACD must be changed. It is often possible to revert to a previously used ACD, but even this is not required, as the installation of any previously unused ACD will simply cause all referenced cache cells to be refreshed on first reference.

LISP/VM binding model

The initial value of the current ACD, CURACD, is a SD associated with the root frame of the stack. This SD, the NILSD acts as an environment containing only the global bindings, as the root frame binds no variables of its own, nether lexical nor fluid.

Whenever a fluid variable is bound the corresponding cache cell is updated. The pre-rebinding contents of the value cell pointer and ACD are saved, on the stack, and restored on exit. The cell is then updated to point to the new binding, with the current ACD stored.

When a free variable is evaluated the corresponding cache cell is examined. If the ACD is the same as the current ACD the value cell pointer is used. Otherwise a search of the stack is made (as in any deep bound model) and the cache cell is updated. In compiled code a second level of cacheing is used. On entry to a section of code which refers to one or more free variable the cache cells for all such are verified, and refreshed if need be. Then their value cell pointers are saved in the dump of the current stack frame, and used directly from there with in the code, with no further verification required. This is a calculated risk, as far as efficiency is concerned. It obviously is advantageous when all free variables are referenced once or more for each invocation of the code. If, however, only a few of the mentioned variables are actually referenced on any specific invocation it is additional overhead. We feel that, on average, the choice was correct.

Since the greatest number of free variables evaluated are those acting as operators in expression, and since these are rarely ever bound on the stack, a special marker is used in the cache cell to indicate a never-stack-bound variable, i.e., one which could not be effected by a context switch.

There are several situations in which the current ACD value must be changed. These are: when resuming a saved state; when applying a funarg or evaluating an expression with respect to an SD; when returning from the application of a funarg or the evaluation of an expression with respect to an SD; when returning from a captured frame which was reached by jaunting.

The new current ACD is, in the first two cases, either the SD in question or a copy of it. In the last two cases the current ACD is set from a slot in the dump being exited, the EXIT field. The EXIT field is used for a number of purposes, including the automatic unbinding of fluid variables during exiting and throwing but the details of the recording of multiple exit procedures need not concern us here.

In describing the rules for establishing the current ACD we will use simplified versions of SDs and stack frames, containing only the components relevant to the current discussion. An SD will be represented as a triple, (frame; current-ACD; in-use-flag). A stack frame will contain an EXIT field. The other components are ignored.

At system initialization time there is a single stack frame, the root frame which has two copies of its dump component, and an SD.

```
NILSD  ==  (root frame; don't care; NIL)
```

The initial value for CURACD is NILSD. As long as processing is of the simple call/return protocol nothing changes. There are four situations of interest.

SD creation.

Assume the current stack frame is F, consisting of head-of-environment H_f and dump D_f. If STATE is called (explicitly or implicitly, e.g. by creating a closure), its value will be an SD, SD_f. A second SD, SD_g, is also created and inserted into the exit field of the retained frame.

```
SD_f     ==  (F; SD_f; NIL)
SD_g     ==  (root frame; SD_g; NIL)
exit(D_f)  ==  SD_f
```

Finally, D_f is copied to the stack frontier as D_f' and execution is continued using that as the current stack frame.

Note that the computation continues under the unchanged current ACD. The entry in the exit field of the retained frame, together with the rule on function return, below, insures that a return from an invocation of the saved state will install a new ACD as current, thus invalidating any cache cells set during the computation following the creation of this SD under the current ACD. Unfortunately, valid cache cells will also be invalidated. The end result is correct, if pessimistic.

Evaluation with respect to an SD (including funarg application).

Assume an expression is to be evaluated with respect to an SD, SD_c, or a funarg containing SD_e is to be applied, from the current stack frame, F_c. There are then two sub-cases.

When the in-use-flag is NIL SD_e may be used as the new CURACD.

```
SD_e   ==   (F_e; SD_x; NIL)

in-use-flag(SD_e) = :T
curacd(SD_e)      = CURACD
CURACD            = SD_e
```

If the in-use-flag is non-NIL, indicating that SD may already be in use as an ACD, the procedure is:

```
SD_e'   =   copy(SD_e)

in-use-flag(SD_e') = :T
curacd(SD_e')      = CURACD
CURACD             = SD_e'
```

Then a new stack frame, F_n, is created with its control chain pointing to F_c and its environment chain to F_e. Finally:

```
exit(F_e)   =   CURACD
```

and either the expression is evaluated or the functional part of the funarg is applied to its arguments.

The setting of the exit field insures that when the evaluation completes the ACD will be restored. The only cache cells effected will be those that have been refreshed during the evaluation and not restored. This can happen due to free variable access, or by jaunting during the evaluation which still allows control to return to this point.

If there is little or no overlap in free/fluid variable usage between the funarg or the evaluation there may be no invalid cache cells produced.

Application of an SD.

Assume that an existing SD, SD_f, is to be applied.

```
SD_f   ==   (F_f; SD_x; in-use-flag)

SD_f'              =  copy(SD&sf)
in-use-flag(SD_f') = :T
curacd(SD_f')      = NILSD
CURACD             = SD_f'
```

The current stack frame, and any of it's ancestors which are not captured by an SD and are at the stack frontier, are deleted and the dump portion of Ff is copied to the frontier and control is passed to it.

The CURACD field in the new SD is set to NILSD, a harmless value. Since the new value of CURACD is a newly created SD we will force all referenced cache cells to be refreshed. The value placed in the exit field of the retained stack frame (Ff) will similarly be used to invalidate cache cells if control returns from it.

Exiting from a frame with exit(F) = SD.

The last case in which CURACD is changed is when returning from a stack frame whose exit field contains an SD. The only times that this condition holds is when one of the previous four situations has occurred.

$$\text{exit}(F_c) == SD$$
$$SD == (F_x;\ SD_p;\ :T)$$

There are then two sub-cases:

$$CURACD == SD$$

in which case:

$$CURACD = SD_p$$
$$\text{in-use-flag}(SD) = NIL$$

but if:

$$CURACD \neg= SD$$

then:

$$CURACD = \text{copy}(SD_p)$$

The occurrence of an SD in the exit field of a stack frame implies a return from an evaluation with respect to some other SD, or a multiple return from a saved state. In either case we must switch the value of the CURACD to one that can not be mistaken for that which we have been running under, or any other that may be extant but not valid at this point in the computation.

In the case of an evaluation we can safely restore a previous value, but if jaunting was involved in control passing to this point we must use a newly created value.

Can we do better?

It would seem that we are creating new (and therefore never-before valid) ACDs at a rather alarming rate. Of course, in "normal" Lisp code (as opposed to pure SCHEME code) this rarely ever actually happens. Even in a Common Lisp implementation it is lexical closures which are used heavily, and they can be dealt with using much weaker machinery.

In fact, there is a variation on this ACD model which adds a small overhead on some validations, but eliminates many of the environment searches. I hope to make that the topic of a following article.

Bibliography

Alberga, C. A., Bosman-Clarke, C., Mikelsons, M., and Van Deusen, M.
Experience with an Uncommon Lisp.
Proceedings of the Lisp and Functional Programming Conference., August 1986.

Bates, Raymond L., Dyer, David, and Koomen, Johannes A. G. M.
Implementation of Interlisp on the VAX.
Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, pages 81-87, Pittsburgh, Pennsylvania, August 1982.

Blair, F.W.
The Definition of LISP1.8 + 0.3i.
IBM Internal Report, 1976..

Bobrow, Daniel G. and Wegbreit, Ben.
A Model and Stack Implementation of Multiple Environments.
Communications ACM, 16(10):591-603, October 1973.

Clinger, William.
The Scheme Environment: Dynamic Variables.
Lisp Pointers, 1(3), August-September 1987.

IBM Corporation.
LISP/370 Program Description/Operations Manual.
IBM Corporation SH20-2076, March 1978.

IBM Corporation.
LISP/VM User's Guide..
IBM Corporation SH20-6477, July 1984.

Padget, Julian and Fitch, John.
Closurize and Concentrate.
Conference Record of the Twelfth Annual ACD Symposium on Principles of Programming Languages, pages 255-263, New Orleans, Louisiana, January 1985.

Teitelman, Warren.
Xerox Palo Alto Research Centers.
INTERLISP Reference Manual, pages 12.1-12.6.
Palo Alto, California. October 1978.

White, Jon L.
LISP/370: A Short Technical Description of the Implementation.
SIGSAM Bulletin, 12(4):23-27, November 1978.

White, Jon L.
Constant Time Interpretation for Shallow-bound Variables in the Presence of Mixed SPECIAL/LOCAL Declarations.
Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, pages 196-200, Pittsburgh, Pennsylvania, August 1982.