Managing Stack Frames in Smalltalk

J. Eliot B. Moss

COINS Technical Report 87-38
April 1987

To appear in the proceedings of the SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques, St. Paul, Minnesota, June, 1987. This is a revised and improved version of Technical Report 86-60.

Abstract. The Smalltalk programming language allows *contexts* (stack frames) to be accessed and manipulated in very general ways. This sometimes requires that contexts be retained even after they have terminated executing, and that they be reclaimed other than by LIFO stack discipline. The authoritative definition of Smalltalk [Goldberg and Robson 83] uses reference counting garbage collection to manage contexts, an approach found to be inadequate in practice [Krasner, et al. 83]. Deutsch and Schiffman have described a technique that uses an actual stack as much as possible [Deutsch and Schiffman 84]. Here we offer a less complex technique that we expect will have lower total overhead and reclaim many frames sooner and more easily. We are implementing our technique as part of a state of the art Smalltalk interpreter. The approach may apply to other languages that allow indefinite lifetimes for execution contexts, be they interpreted or compiled.

1 The Problem

The Smalltalk language presents many implementation challenges, with efficient call and return heading the list. The reason call and return are hard to implement efficiently is that contexts (the Smalltalk terminology for stack frames) are not used in a strictly stack-like way. They can be treated as general objects, stored, and perhaps modified, by user code. Contexts must be retained until they are no longer accessible. Yet, for performance we need to be able to allocate contexts very rapidly (it would be best to be competitive with stack based languages such as Pascal) and reclaim them efficiently, too. In this section we provide background on relevant features of Smalltalk.

1.1 Smalltalk Method Contexts

In Smalltalk a procedure is called a *method.* Method invocation takes place by *sending* a *message* to an object, called the *receiver* of the message. The message consists of a *selector* and zero or more *arguments.* The selector indicates the name of the desired operation; based on the *class* of the receiver (i.e., its type), the operation name is looked up and an appropriate piece of code is found, namely a method. Neglecting the dynamic binding that results from the lookup of the method based on its name, a send is very similar to a traditional procedure call. The receiver and arguments play the role of parameters to the procedure, and the method is the procedure itself.

When a send occurs, a *method context* is created. The context contains the parameters of the call, space for local variables (called *temporaries*) and working stack, and a reference to the previous context (called the *sender*). When execution in a context is suspended (e.g., when it does a send), its stack pointer and instruction pointer for resumption are stored in the context. Thus a context itself contains all state necessary for its own resumption. In that sense, a context is similar to a continuation.

Unlike Pascal, where procedures may have procedures defined within their scope, resulting in many lexical levels and requiring some means of accessing them, Smalltalk methods are essentially all global, and thus need no access to enclosing scopes. There are global variable of several kinds, but they introduce no difficulties.

1.2 Smalltalk Blocks and Block Contexts

If Smalltalk had only method contexts, and did only sends and returns, then it would obey stack discipline and context management would be easy. However, inside a method one may define nested pieces of code called *blocks.* Typical examples are the *then* or *else* arm of a conditional, and the body of a loop. However, when a block is evaluated, the code is not evaluated immediately. Rather, an object called a *block context* is constructed. When the block context is sent the particular message `value`, it runs the code, and returns whatever value is produced by that code. An example should make this more clear:

```
b ifFalse: [y ← 0] ifTrue: [y ← 1]
```

Here the (object bound to the) variable `b` is sent the selector `ifFalse:ifTrue:`, with the two block contexts as arguments. The objects `true` and `false` react to `ifFalse:ifTrue:` in different ways: `true` sends `value` to its second argument (the `ifTrue:` block), and `false` sends `value` to the other. If `b` is not a boolean, a run time error will result.[1]

It should be easy to see how this notion of blocks of code allows one to build a variety of control structures directly in the language. However, block contexts present some implementation problems. First, they are lexically contained in a method, and hence they refer to the method context in which they were created, so as to allow access to the method's arguments, temporaries, etc. Since blocks are objects, they can be stored at will into other objects, and the references can persist indefinitely. The implication is that not only the block context, but also the method context to which it refers (called the block context's *home* context), may need to be retained.

A block context can be sent the `value` message many times; this is used to implement loops. One can readily achieve, and exceed, the power of iterators as offered by CLU [Liskov, et al. 1977, Liskov, et al. 1981] and Trellis/Owl [O'Brien 85, Schaffert et al. 85].

Blocks correspond closely to lexically nested procedures in Pascal, and to nested lambda expressions in Lisp. In fact, the implementation problems posed by blocks are not unlike the so-called funarg problem in Lisp. However, there are some minor differences that are slightly helpful. First, a block's local variables are actually stored in the home context. This means that although blocks may be nested within blocks, no display or static chain is required (beyond a reference to the home context) since all local variables are in the home context.[2] Second, once a method context has returned, unlike a block context, it cannot be re-entered. While we might debate whether this is the best language design, the fact remains that it is how Smalltalk is currently defined.

1.3 Other Features

A few other features of Smalltalk are of relevance. First, not only can blocks be created, passed around, and stored, but an executing context can actually obtain a pointer to

[1]Actually, the Smalltalk compiler optimizes a number of these blocks away, turning them into the usual conditional and looping code; but the user is still allowed to manipulate blocks with the full power described, so the system must support it.

[2]This aspect of the language may change. However, it does not strongly affect the allocation/deallocation issues we are addressing here.

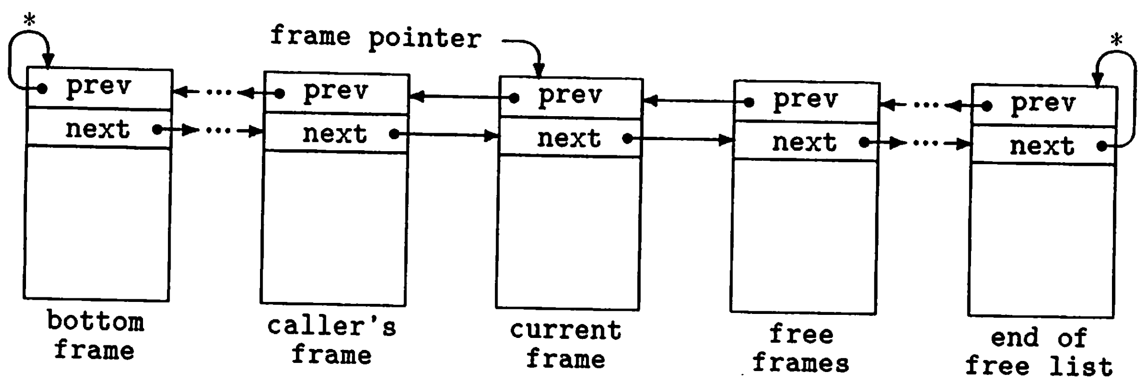

Figure 1: Basic Technique for Chaining Frames Together.

itself. In fact, block contexts are created by sending blockcopy to the current context. Second, any field of a context can be examined and modified by sending a message to the context. This ability is used to write the Smalltalk debugger in Smalltalk, but is available for general use as well. Finally, Smalltalk supports multiple threads of control. The threads are lightweight processes running in the same Smalltalk universe. However, only one runs at a time (multiprogramming, not multiprocessing). Still, we must be able to handle process swaps, and since processes are used to respond to interrupts, swaps must be reasonably fast and efficient.

2 Our Solution

We now describe our context management scheme. In brief, we use a doubly linked list in stack-like fashion most of the time. When frames need to be retained, we unlink them and change various fields. The details will be presented in stages. Finally, we discuss some techniques that might be employed to reclaim some frames sooner than the garbage collector will find them.

2.1 The Basic Idea

In the beginning, we pre-allocate some number of frames (analogous to initial allocation of stack space for a stack based language), and build them into a doubly linked list. We will call the links prev and next. A frame's prev link normally points to its sender (i.e., the frame to which it should return when done) and its next link points to a currently unused frame. Thus, when a send occurs, we use the next frame for the new method context. When we return, we simply follow prev back to our caller and resume executing it. This is illustrated in Figure 1. Obviously our linked list is not infinite, and we must somehow deal with its two ends. This is indicated in the figure; the details will be explained later.

Since frames are pre-allocated, they each have the same fixed size. This is possible in Smalltalk because the language limits the size of a context. Even if this were not the case, one could probably dynamically allocate part of frames large than some threshold, and use the scheme we are proposing except in the rare case of unusually large frames. It is important to understand that the storage part of variable in a frame is always in a heap (they are full-fledged, separate objects), so the size of frame is proportional only to the number of variables, not to the amount of data to which those variables refer.

Another point we should make now is that although Smalltalk defines the format of context objects in detail, our frames have a different format, for reasons of efficiency. To get away with this, we must provide Smalltalk programmers the illusion of the context objects they expect. We do so by special handling in the interpreter of references to and modifications of the fields of context objects. This requires writing a little bit of Smalltalk code, and is not strictly standard, but results in only a minor change to the Smalltalk system. The difference in format is why we refer to the objects in our design as *frames* rather than contexts.

2.2 Kinds of Frames

The frames mentioned above are called *volatile* (if in use), and *free* (if not in use). Suppose frame *a* has frame *b* as its `next`, and we are executing in frame *a*. Frame *a* (and its `prev`, etc.) is volatile, while frame *b* (and its `next`, etc.) are free. When we do a send, *b* becomes volatile; when *b* returns it is made free again. We will always attach the list of free frames to the currently executing frame.

It is quite important to realize that the normal case does not involve any special manipulations to achieve the allocation and freeing: it is following the pointers back and forth that accomplishes the implicit allocation and reclamation of volatile frames. The normal case for send and return is very efficient: one instruction on a typical computer. This compares favorably with frame pointer save/restore overhead for contiguous stacks. The full instruction sequences for send and return can also be designed to come out within an instruction or two of what would be required in a contiguously allocated stack. Since there are issues other than frame allocation and reclamation involved when designing calling sequences, we defer detailed examination of this issue to a later section.

When a frame may need to be retained, it is converted from volatile to *stable* form. The exact manipulation will be described later. However, it is useful to know that there is a flag or condition in a frame that allows us to distinguish stable and volatile frames upon inspection. A typical implementation would just use a flag bit.

Stable frames have their `prev` link broken, which enables us to avoid reclaiming them

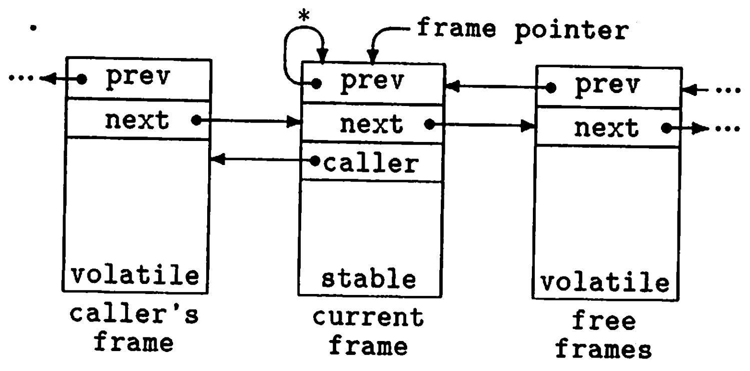

Figure 2: Chaining for stable frames.

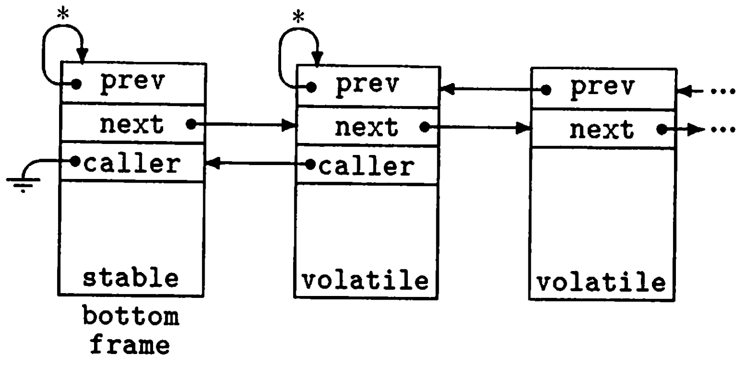

Figure 3: The base of a process stack.

when they return. The old prev link information is stored in the `caller` field, a field initially set to null in volatile frames. The prev link is broken in a very particular way – it points back to the stable frame itself, is detectably different, but recoverable. The linking of stable frames is illustrated in Figure 2.

The base frame of a stack also has its prev link broken, but its `caller` will be null, since it has no place to which to return. The system can detect this as part of its special handling of broken prev links. While a base frame theoretically might be volatile, the way processes are created pretty much dictates that base frames will be stable, since the initial process state object has a pointer to the base frame. This can be seen in Figure 3.

The specific trick we have used to indicate broken pointers is to *negate* the original pointer. This trick allows us to do the send or return action on any frame, detect the broken link, and handle the broken link case specially, and do all of it very efficiently. Assuming the pointer to the current frame is in `fpReg`, the following C code illustrates the trick:

```
# define broken(x) (x) < 0
```

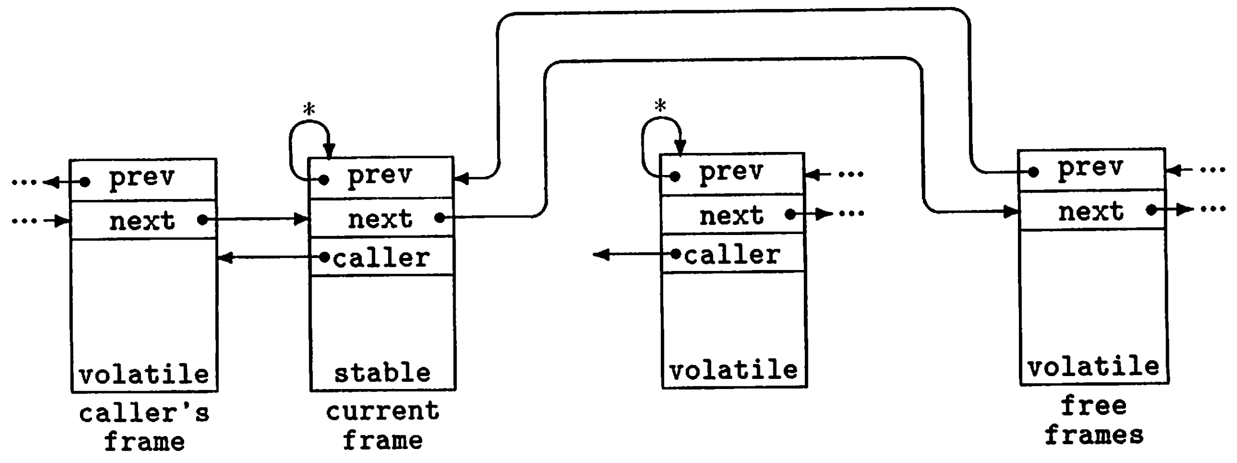

Figure 4: Situation after stable frame is forcibly re-entered.

```
# define recover(x) x =- x

if (broken(fpReg = fpReg->prev)) {
   /* prev was broken */
   recover(fpReg);
   ... special case code
   }
```

On the VAX[3], this results in something like the following code:

```
    movl 8(fpReg), fpReg
    bgeq ok
    mnegl fpReg, fpReg
    ... special case code
ok: ... rejoin normal code
```

The point is that the normal case has been slowed down only by the check for less than zero. We will use the same technique when we introduce broken next links later. Clearly there are variety of other ways to accomplish the same end; this particular trick simply works out nicely on the VAX.

2.3 Possible Link States

Normally, a volatile frame's next and prev connect it to other volatile frames, allowing efficient allocation and reclamation. However, if a volatile frame's sender is (or becomes) stable, and the stable frame is re-entered other than by a return (e.g., re-entry is forced by the debugger), then the stable frame's next and the volatile frame's prev must be broken. This is to insure that the volatile frame is not clobbered if we call out of the stable frame,

[3]VAX is a registered trademark of Digital Equipment Corporation

```
if (broken(fpReg = fpReg->prev)) {
    /* return from stable frame, or volatile with re-entered caller */
    recover(fpReg);

    /* check that caller exists, and that it has not returned */
    if (nocaller(fpReg) || returned(fpReg->caller)) {
        ... cause "cannot return" error
        }

    /* determine free list and do fix up actions */
    target = fpReg->caller;
    free   = fpReg;
    if (stable(fpReg)) {
        free = fpReg->next;

        /* mark frame as terminated */
        makereturned(fpReg);
        }
    else
        /* restore free frame to volatile format */
        setnocaller(free);

    /* attach free list and set fpReg */
    target->next = free;
    fpReg = free->prev = target;
    }
... restore stack pointer, instruction pointer, etc.
```

Figure 5: C code for returning.

and also to allow us to attach the free list to the stable frame. In this case the volatile frame's `caller` will point to the stable frame, just as `caller` in a stable frame does. This is shown in Figure 4. For similar reasons, if a volatile frame is at the end of the free list, its `next` link is broken. This has already been illustrated in Figure 1.

A stable frame always has a broken `prev` link. Its `caller` link contains the caller information, until the stable frame returns. In Smalltalk, when a frame finishes, it stores nil in its caller and instruction pointer fields. In our design, volatile frames are always reclaimable (and reclaimed) when we return from them, and the storing of nil is not necessary. However, when a stable frame returns, we destroy the `caller` field and store a special value there, indicating that the frame has returned.

2.4 Return Actions

Figure 5 describes the frame manipulations performed to effect a return. The special

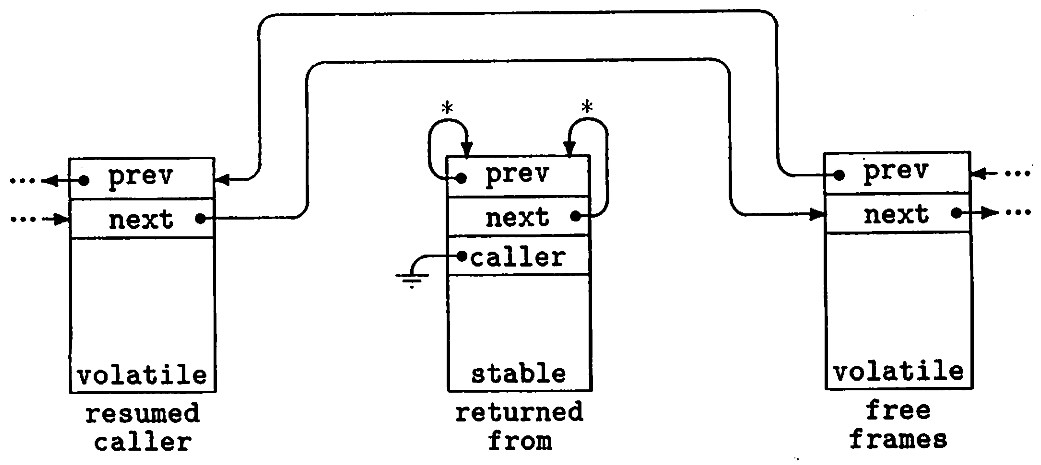

Figure 6: After a stable frame has returned.

case code is the interesting part. It is executed when returning from a stable frame, or from a volatile frame whose caller was forcibly re-entered. We cannot perform the return if there is no caller (this handles frames at the base of the stack, etc.), or if the caller has already returned (such a frame is probably a relic of a frame that was forcibly re-entered). If we can return, there are two cases: (1) the current frame is stable, in which case we cannot free it, and must mark it as returned-from; and (2) the current frame is volatile, in which case we need to clear out its caller slot, since it will now be free. In either case we need to attach the free list to the frame to which we are returning. The situation after a stable frame has return is shown in Figure 6.

2.5 Send Actions

Sending is straightforward: we need only detect that we are out of frames:

```
if (broken(fpReg = fpReg->next)) {
   /* end of free list */
   recover(fpReg);
   ... get more frames or give up ...
   }
... save old frame info
... set up new frame, etc.
```

2.6 When to Stabilize

There are three ways in which externally visible pointers to frames arise. The first is via the pushThisContext instruction, which is how the user obtains a reference to the running context. It is also used in the creation of block contexts. In either case, the frame executing

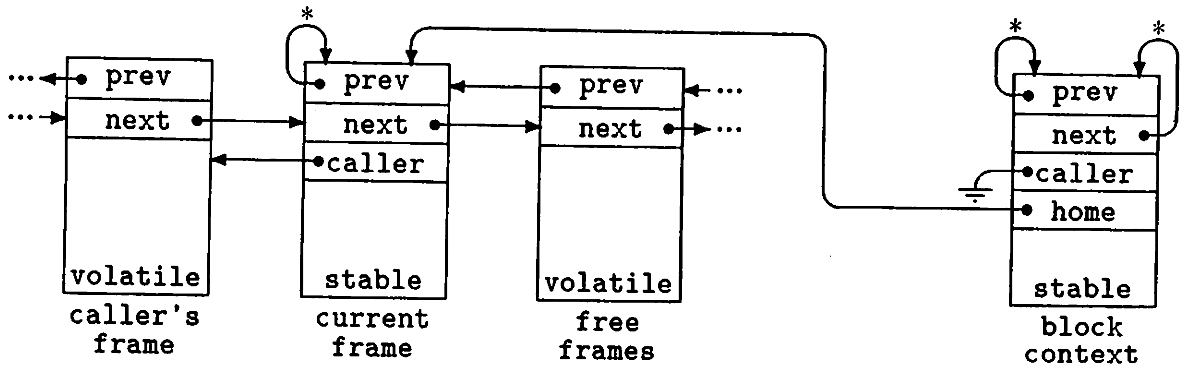

Figure 7: Current frame creates a block context.

pushThisContext should be stabilized. Here is the code to do it:

```
if (unbroken(fpReg->prev)) {
   /* has proper prev link */
   fpReg->caller = fpReg->prev;
   fpReg->prev   = -fpReg;
   }
/* mark as stable frame */
markstable(fpReg);
```

The second way in which pointers arise is when block contexts are created. When that happens, we are always in a stable frame (we may have just stabilized it). We simply grab a frame from the free list and make it stable. This is illustrated in Figure 7. Here is the code for doing it:

```
/* get a frame */
if (broken(new = fpReg->next)) {
   ... out of frames
   }
/* unchain it */
fpReg->next = new->next;
if (unbroken(fpReg->next = new->next))
  fpReg->next->prev = fpReg;
/* make it stable */
markstable(new);
new->prev = -new;
setnocaller(new);
new->home = fpReg;
... continue initializing frame
```

The situation that results when the block is later invoked is shown in Figure 8 (see also the code below). Note that by tracing back the caller chain (through caller of stable

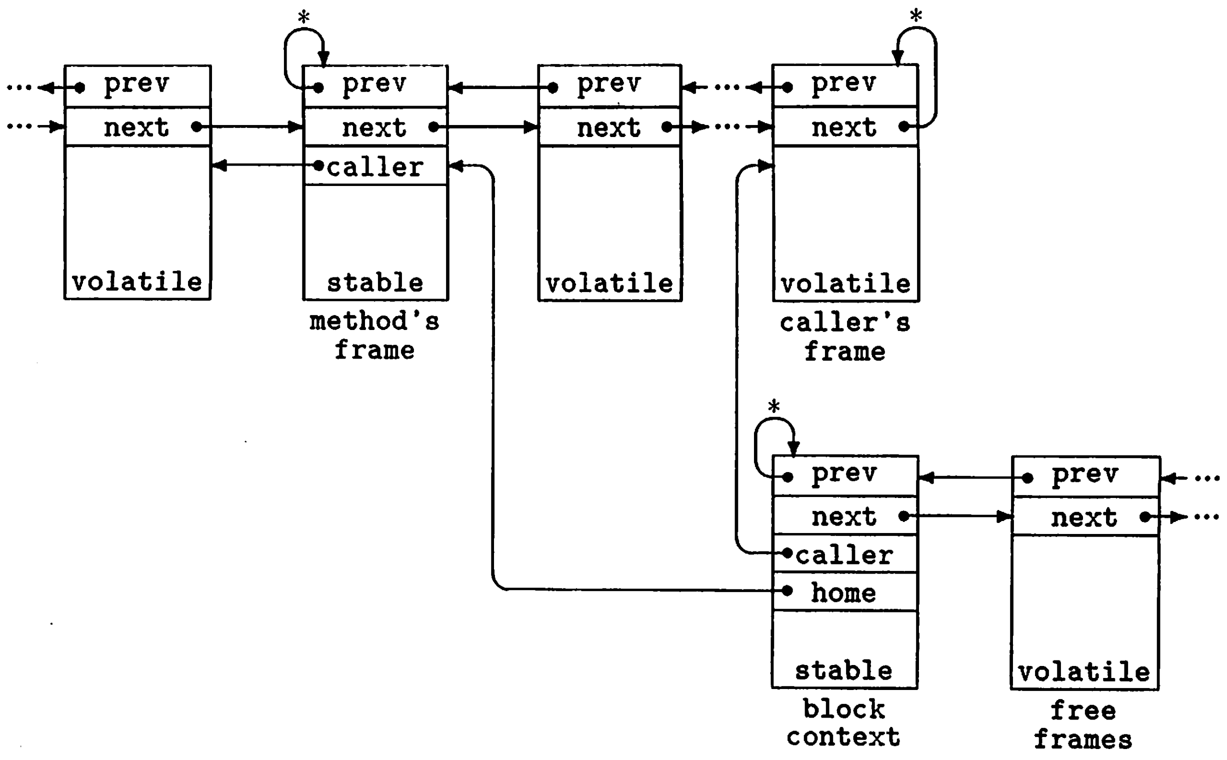

Figure 8: Block during execution.

frames and prev of volatile ones), we should eventually get back to the home context.[4] If in executing the block, the code returns from the method (a "non-local" return), then the volatile frames up the caller chain are reclaimable. This is discussed further later.

The final way in which pointers arise is when they are fetched out of frames. This can be detected by a combination of code in the interpreter and the previously mentioned changes to the Smalltalk code. The frame whose pointer is obtained must be stabilized. The code is like that for stabilizing the current frame, so we do not present it.

The only other case we must handle is entering a stable frame other than by returning into it; this includes sending value to a block context. Assume a pointer to the frame to be resumed is in new; here is the necessary code:

```
if (unbroken(oldnext = new->next) &&
     volatile(oldnext)) {
    /* note forcible re-entry */
    oldnext->caller =  oldnext->prev;
    oldnext->prev   = -oldnext;
    }
/* attach free list */
```

[4]If the block was invoked through a stored reference, this might not be the case, though it would be unusual. Our design would not break, but such possibilities are not shown in Figure 8.

```
if (unbroken(new->next = fpReg->next))
   new->next->prev = new;
```

2.7 Reclaiming Some Frames Quickly

The scheme just described works, and reclaims perhaps 90% of frames allocated, but stable frames are reclaimed only via some garbage collection technique. Also, some volatile frames will not be reclaimed. This comes about when a block context does a "return from method", as previously mentioned. What happens is that control is transferred to the home context's caller, and any volatile frames back up the block context's `caller` chain are lost. In any case, it is conceivable that we could do better by detecting some of the most common cases and reclaiming frames that we can. In this section we mention some possible extensions to our scheme that might be useful, but we do not present detailed code.

An obvious special case is when a block does a "return from method". We can tack any volatile frames back up block's `caller` chain onto the free list and reclaim then immediately.

Another normal case we would like to handle better is that of executing a method which uses a block as a loop body or conditional arm. The Smalltalk compiler optimizes away the blocks written for *if/then/else* and *while* constructs, which helps a great deal. Still, there are other sorts of loops and conditionals that could benefit from special treatment in reclamation.

A fact we can use to our advantage is that if no pointer to a block context has been explicitly stored or assigned to any variable, then the block context can be freed when its home context returns. Detecting stores into the heap and into certain fields of contexts fits in with the interpreter as previously envisioned. However, suppose a block b_1 receives block b_2 as one of its arguments, and then stores b_2 into one of b_1's home context's temporaries. If b_1's home outlives b_2's, then we cannot reclaim b_2. Similarly, if a block's home context returns the block itself as its value, the block has escaped the dynamic scope of its home. Thus, to reclaim blocks when their home returns, we need to detect those blocks that might outlive their home. The necessary checks are possible, and not very complicated. The question is whether or not the improvement is worth the overhead. We intend to investigate this issue.

Another case that may deserve special treatment is the heap references to contexts that arise from process swapping. In this case, a `Process` object refers to the context on the top of the process' stack, so that the process can be later resumed. These references to contexts typically are not referred to in any other way. Hence, distinguishing heap

Access arguments via:	`+n(fp)` (stack grows "downwards")
Access locals via:	`-n(fp)`

Send action	Code	Return action	Code
save fp	`pushl fp`	restore sp	`movl fp, sp`
new fp	`movl sp, fp`	restore fp	`movl (sp)+, fp`
alloc locals	`subl2 #n, sp`	discard args	`addl2 #n, sp`
Cost =	3 instructions + 1 memory reference	Cost =	3 instructions + 1 memory reference
Total cost = 6 instructions + 2 memory references			

Figure 9: Calling sequence for contiguous stack.

references in `Process` objects from other heap references may be worthwhile.

Finally, although many frames are not immediately reclaimable, few have heap references to them. Therefore, if we put contexts in a separate area, and we do recursive tracing on that area using as roots those contexts that have been referred to from the heap at some time, then we will likely reclaim many contexts without garbage collecting the main heap.

3 Previous Work

We now briefly compare our scheme with some other published techniques.

3.1 Bobrow and Wegbreit

A stack frame management scheme was described in [Bobrow and Wegbreit 73] some time ago; the method has sometimes gone by the name "spaghetti stacks". Spaghetti stacks use reference counts for reclamation, but achieve contiguous stack frame allocation in the normal cases. Our scheme should perform better in terms of normal case allocation and reclamation, since we avoid reference count overhead, and our normal case code is at most a few instructions. The amount and contiguity of memory for stack frames would not seem to be an issue in modern workstations, since the total amount of memory involved is but a small fraction of that used by the Smalltalk system as a whole.

3.2 Deutsch and Schiffman

The PS implementation of Smalltalk, described in [Deutsch and Schiffman 84], uses a true contiguous stack. It uses the normal procedure call and stack frame allocation and reclamation technique offered by the 68000 series microprocessor instruction set. Comparison is difficult, since what matters is the entire cost of the send/return mechanism. In

Access arguments via:	`-n(fp)`	(stack grows "downwards")
Access locals via:	`-n(fp)`	
Access new args via:	`-n(sp)`	(see text)

Send action	Code	Return action	Code
set up fp	`movl sp, fp`	restore sp	`movl fp, sp`
set up sp	`movl next(fp), sp`	restore fp	`movl prev(fp), fp`
Cost =	2 instructions + 1 memory reference	Cost =	2 instructions + 1 memory reference
Total cost = 2 instructions + 2 memory references			

Figure 10: Calling sequence for our scheme.

particular, we must consider how arguments are accessed, etc. A simplified analysis (we are assuming the all other costs are equivalent) can be seen in Figure 9 and Figure 10. The contiguous allocation code sequences are as for the VAX; the 68000 has `link` and `unlink` instructions that reduce the number of instructions necessary (but not the extra memory references). The VAX `call` instructions are much heavier weight and would be a poor choice for this application. Both systems must check for running off the end of the stack, etc. – the costs are comparable.

To obtain the efficiency offered in Figure 10, we must "push" (i.e., store) arguments for the next call in the next frame. This prevents our having to move them there later. An alternative we considered was leaving the arguments in the caller's frame. This works (until the frame is stabilized), but requires an extra pointer to be maintained, at a cost of two instructions, and possible two memory references, per send/return pair. It is also more complex. The scheme in which argument are built in the new frame has a minor drawback: intermediate results cannot be stored on the stack, but must be saved in local variables, and then moved back to the "stack" right before use. The added overhead is at most a store/fetch pair of references, which is what it would have cost to maintain the separate pointer to the caller's frame. If that store/fetch pair is though of as being charged to the call that created the intermediate result, then, since not all results need be saved that way, we come out ahead. Implementing this scheme will necessitate changing the compiler.

The main difficulty with the PS scheme is that it is quite complex, maintaining frames in three formats: volatile, hybrid, and stable.[5] The conversions and correspondences between these formats are not trivial. Further, stacks must be allocated with some kind of limit, and PS must do a limit check (similar to our check for no more free frames), and chain multiple stack segments together when one overflows. Dealing with stack segment chaining

[5]We took our volatile/stable terminology from PS.

adds considerable additional complexity.

In sum, our scheme appears to be considerably simpler, and of comparable or even lower cost in the normal case, as well as being less costly in the special cases (we never need to move frame contents to stabilize, etc.).

3.3 McDermott

In [McDermott 80] there is a design of a frame management system for Scheme. That design uses contiguous stacks, moving things to a heap only when necessary. Some of McDermott's analysis of the various cases, especially ways to reclaim frames that our design stabilizes, should be extended to our system. However, a number of the arguments made about PS also apply here - we do not have to move things around, our cost is comparable to contiguous allocation, etc. One interesting aspect is that we assume fixed size frames are adequate (they are in Smalltalk as it currently exists), which may not be reasonable for LISP. Still, fixed size frames can probably be made to work even if there are no limits on the numbers of variables, etc. One just allocates parts of large frames as separate objects, and pays an extra penalty when large frames are used. In sum, our techniques, when applicable, will likely out-perform McDermott's basic approach. But his analysis of a number of situations should be applied to our scheme to improve reclamation of block contexts, etc. It is probably not urgent to do this, since PS gets by without applying such techniques.

4 Conclusions

We have presented a scheme for managing stack frames in Smalltalk which should be competitive in performance with the best known techniques, yet which is simpler. We have also described some extensions to reclaim some frames faster - McDermott's ideas should be applied here. The scheme is being implemented in a Smalltalk interpreter currently under construction, but no implementation results are available as of this writing.

References

[Bobrow and Wegbreit 73] Daniel G. Bobrow and Ben Wegbreit, "A Model and Stack Implementation of Multiple Environments", *Communications of the ACM*, Volume 16, Number 10, October 1973, pp. 591–603.

[Deutsch and Schiffman 84] L. Peter Deutsch and Allan M. Schiffman, "Efficient Implementation of the Smalltalk-80 System", *Conference Record of the Eleventh*

Annual ACM Symposium on Principles of Programming Languages, January 1984, pp. 297–302.

[Goldberg and Robson 83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Krasner, et al. 83] Glenn Krasner, ed., *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983.

[Liskov, et al. 1977] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM*, Volume 20, Number 8, August 1977, pp. 564–576.

[Liskov, et al. 1981] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder, *CLU Reference Manual*, Springer-Verlag, 1981.

[McDermott 80] Drew McDermott, "An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-Scoped LISP", *Conference Record of the 1980 LISP Conference*, August 1980, pp. 154–162.

[O'Brien 85] Patrick O'Brien, "Trellis Object-Based Environment: Language Tutorial", Version 1.1, Eastern Research Laboratory, Digital Equipment Corporation, Technical Report 373, November 1985.

[Schaffert et al. 85] Craig Schaffert, Topher Cooper, Carrie Wilpolt, "Trellis Object-Based Environment: Language Reference Manual", Version 1.1, Eastern Research Laboratory, Digital Equipment Corporation, Technical Report 372, November 1985.