MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 556 July 16, 1980

Phantom Stacks
If you look too hard, they aren't there

by
Richard M. Stallman

Abstract:

A stack is a very efficient way of allocating and deallocating memory, but it works only with a restricted pattern of usage. Garbage collection is completely flexible but comparatively costly. The implementation of powerful control structures naturally uses memory which usually fits in with stack allocation but must have the flexibility to do otherwise from time to time. How can we manage memory which only once in a while violates stack restrictions, without paying a price the rest of the time? This paper provides an extremely simple way of doing so, in which only the part of the system which actually uses the stack needs to know anything about the stack. We call them Phantom Stacks because they are liable to vanish if subjected to close scrutiny. Phantom Stacks will be used in the next version of the Artificial Intelligence Lab's Scheme microprocessor chip.

Keywords: Co-routines, Funarg Problem, Spaghetti Stacks

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's 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-0643.

Background

When allocating memory space which may be needed for unanticipated reasons for an unpredictably long time, it must be taken from a dynamically allocated area or "heap". In a Lisp system, with its cyclic pointers, this means that a garbage collector is needed to reclaim the space.

However, when it can be guaranteed that the most recently allocated memory will remain in use for the shortest time, the memory can be allocated in a stack instead. The stack functions as a strategy for explicit reclamation which eliminates the need for slow garbage collection. However, the stack can only safely be used for memory whose usage absolutely always follows this pattern. For memory which usually but *not quite always* follows a stack discipline, the stack cannot be used at all.

Because of the speed of stacks, Lisp interpreters traditionally use them internally for such things as variable bindings and control information. However, this limits the flexibility of the language, because interesting forms of control structure would prevent the stack from being used. For example, the current variable bindings obey a stack discipline for ordinary function calling, but the occasional use of FUNCTION causes all the variable bindings which then exist to have to be saved for unpredictably long. If FUNCTION is to work properly, the variable bindings cannot be kept in a stack [See Function]. The classical "solution" to this problem is to store them in lists in the heap; but this makes code run slower all the time, whether FUNCTION is used or not. Having FUNCTION in the system costs even when it is not used. As a result, most practical Lisp systems today choose to leave it out.

Lisp dialects such as Conniver and Scheme [see Scheme] provide the ability to save the entire control state of the system, to be resumed later. This provides the facility of co-routining, which is extremely powerful. In the AI Lab's Scheme microprocessor chip [SChip], an interrupt will simply cause a co-routine call to a pre-specified *catch tag* (saved control state), creating another catch tag to save the state that existed before the interrupt. But the Scheme catch tag does the same thing to the control state information that the use of FUNCTION does to Lisp 1.5's binding environment: it prevents a stack from being used. As a result, these languages are inevitably slower than ordinary Lisp.

So, the obvious problem is: how can the flexibility of FUNCTION and co-routines be provided without costing the user who doesn't use them? In other words, how can one manage memory whose usage pattern is almost stacklike, so as to get most of the advantages of a stack? And can this be done without increasing the complexity of the rest of the system? This is the problem solved in this paper.

Previous Work

A widely known way for managing almost-stacklike memory other than a general heap is the "spaghetti stack" technique, developed in 1972 by Bobrow and Wegbreit [BW]. Here, the almost-stacklike memory resides normally in an ordinary stack. A sort of reference count is used to tell whether an element of the stack might possibly be needed after the time at which it would ordinarily be popped. If that is so, then when it is popped a copy is actually made elsewhere (in the heap). This requires being able to keep track of all pointers to objects on the stack. These can't be ordinary Lisp objects that anyone can point to; pointers to the stack are restricted to living inside special *environment pointer* objects which so that the system can keep track of them.

Spaghetti stacks successfully solve the problem of getting the speed of a stack when the flexibility is not used, but at a great cost in complexity which has discouraged most people from trying to implement them. They also have the specific problem of being unable to save variable binding information without saving control information as well.

The AI Lab Lisp machine [LM] system takes another tack, offering multiple "stack groups" which are entirely disjoint stacks. The program switches from one stack group to another on explicit request. Allocation within a given stack group is completely stacklike. The intention of stack groups is to allow the most useful non-stacklike operations (such as straightforward coroutines) while losing no efficiency in the normal case. They succeed in the latter goal, but it is not clear that they succeed in the former; also, the overhead of creating a stack group is significant, which discourages their use for many possible applications.

A different approach to optimizating the use of almost-stacks is to push fewer things. Simple recursive interpreters do many pushes saving variables in case they will be altered by deeper levels of recursion. Usually they are not altered and saving them was unnecessary, in hindsight. Steele and Sussman [Dream] have developed a technique of on-the-fly optimization which avoids these unnecessary stack operations. Their technique can be used together with the methods of this paper.

It is rumored that a technique similar to the one in this paper, though with different philosophy, may be used internally by Simula, but we have been unable to find out for sure.

Phantom Stacks

In nearly all Lisp systems, any stack which is used is an *object.* It may not be visible to the Lisp user as a Lisp object, but internal to the system it acts the same way: an area of memory *is* a stack; it has "stack" written all over it for any part of the system which cares to look (and many parts had *better* look).

The key to reducing the impact of a stack on the complexity of the system is to realize that a stack really is not a sort of object that lives in memory, but a way of allocating memory among other objects. It need not have any existence, except in the "mind" of some part of the system which is allocating memory, for some length of time. If the data which is stored in the stack looks like ordinary Lisp objects, then most of the Lisp system will be able to deal with it properly without having to be aware that there is a stack.

One consequence of making the stack not recognizable as a stack is that the garbage collector won't know it is a stack. It will compactify, reorder or reallocate it willy nilly. But the stack's contents, valid accessible Lisp objects, will be preserved in meaning by the garbage collection, just like all such Lisp objects. So no harm is done, as long as we realize that the stack isn't a stack any more. The area of memory which used to be a stack is now part of the heap.

In addition, since the stack's contents look like part of the heap, we have the option of deciding at any time to call it part of the heap. We can use the stack area as a stack for as long as it is convenient. If we run into a snag and can't see how to keep using it as a stack, we can do what the CIA does when its agent is caught: we can pretend it was never a stack at all, and say that all or part of what used to be a stack is now just part of the heap. This is how we deal with the embarassing infrequent call to FUNCTION that requires us to save all the data in the stack indefinitely.

Implementation

We must require that the system use a compactifying or copying garbage collector, so that ordinary consing can just increment a free pointer to get the next available cell. This is so we can allocate memory in large chunks and not just scattered words, when we like.

Now let's assume that one component internal to the Lisp system has a list to keep track of which is nearly always used as a stack. Calling the internal variable which holds the list L, this means that at some times this part of the Lisp system will do

```
(SETQ L (CONS ... L))
```

and at some times it will do

```
(SETQ L (CDR L))
```

and also assume that it will not hand out the value of L, or any number of cdr's of L, to the rest of the Lisp system or to the user's Lisp program except at clearly identifiable instants.

To avoid having to do conses in the heap, the subsystem can allocate a large block of contiguous memory and use it as a stack. The stack is allocated as far as most of the Lisp system can see, since the free pointer CONS uses has been bumped past it, and CONS will not try to give it out again. However, in another sense, it is yet to be allocated, because its individual words will be allocated to cells of L, reclaimed, and allocated again.

The detailed algorithms will be written in a pseudo-Lisp with no distinction made between Lisp object pointers and integers. We won't worry about types of objects, or about any Lisp objects which aren't pointers. They would just obscure the issue. For variables, we have L, of course, and then BEGA and ENDA which point to the bounds of the stack (ENDA is the first location *not* included), and P which points at the next location to push into. P equals BEGA when the stack is empty and equals ENDA when the stack is full. We'll assume that a Lisp cell occupies one location, for simplicity. We need to have a state in which there is no stack. BEGA=0 and ENDA=0 will do for that.

First, we need to be able to create a stack. Let's assume that FREE is the address of the beginning of the free area of memory; the place where CONS would allocate the next cell.

```
(DEFUN MAKESTACK ()
   (SETQ BEGA FREE)
   (SETQ P BEGA)
   (SETQ FREE (+ FREE SIZE))
   (SETQ ENDA FREE))
```

We have to be able to push onto L. We do this by always using the next free location in the stack.

```
(DEFUN PUSH (X)
   (AND (= P ENDA) (MAKESTACK))
   (STORE-MEMORY X L P)
```

```
        (SETQ L P)
        (SETQ P (1+ P)))
```

(STORE-MEMORY CAR CDR ADDR) stores CAR in the car, and CDR in the cdr, of the cell at address ADDR. Note that if the stack is full we make a new one and forget all about the old one. If we really do push and pop up and down over a range larger than one of our stacks, this will mean some wasteful garbage collection, since we can only automatically reclaim cells which are part of the most recent stack, and not cells which are parts of previous stacks. An actual system might try to allocate a larger stack in this situation.

We also need to pop. When popping, we assume that it is safe to reclaim the cell that we are popping off, provided it is in the stack. But we don't assume that it is inside the stack. It must have been inside the stack when we pushed it, but it might no longer be, if a pointer to the list L was handed out since then.

```
(DEFUN POP ()
   (AND (< (CDR L) ENDA)
        (>= (CDR L) BEGA)
        (SETQ P (CDR L)))
   (SETQ L (CDR L)))
```

Finally, we need a function to disavow part of the stack if necessary whenever a pointer to L, or some tail of L, is handed out to the outside world. For a stack used to hold variable bindings, this corresponds to a call to FUNCTION. Note that dispel does nothing when given a pointer that is not inside the stack. This can happen, since the back part of L might live in an area which is not stack any more.

```
(DEFUN DISPEL (TAIL)
   (AND (< TAIL ENDA)
        (>= TAIL BEGA)
        (SETQ BEGA TAIL))
   (AND (> BEGA P)
        (SETQ P BEGA)))
```

This causes whatever part of the stack might contain the part of L we have handed out, to stop being part of the stack. Note that if the stack is full and TAIL equals L, then the stack will become zero size. The next push will make a new stack.

But one thing is left: garbage collection. After a garbage collection, the data which used to reside between BEGA and ENDA will be somewhere else, and that

area will be jammed full of other Lisp cells (or perhaps free). In any case, we have to stop trying to use it as a stack.

```
(DEFUN NOTEGC ()
   (SETQ BEGA 0)
   (SETQ ENDA 0))
```

Each component of the system that might be using a stack internally needs to be told to forget it, after the garbage collection is done.

Switching Between Stacks

If we use a single phantom stack to hold control information, then an interrupt, which must save the control state, will have to cause the stack to be forgotten. If interrupts happen every 60th of a second, to switch between processes all executing in Lisp, this might be an undesirable inefficiency. The ideal thing to do here is to have several stacks and switch between them, just as the Lisp machine switches between stack groups [LM]. Not all times when a control state is saved ought to be handled this way, but it would be nice if the system could decide freely how to treat the stack each time a control state is saved.

We can extend the phantom stack system to handle switching between stacks easily if we add to the system a counter, called GCCOUNT, which is incremented every time there is a garbage collection. If the garbage collector has no idea where the actual stack in use is, it certainly doesn't know about all the saved pointers to other stacks, so a garbage collection will invalidate them all. We manage this by saving the count of garbage collections whenever we save a description of a stack. When we reselect a stack from a description, we have to verify that the stack hasn't been collected since the description was saved.

Conveniently, having multiple stacks doesn't have any effect on the operations defined in the previous section. All it does is introduce two new operations: SAVESTACK and RESTORESTACK. SAVESTACK returns a list which describes the current stack. RESTORESTACK is given such a list and starts using that stack again, but only if no garbage collection has happened in between.

```
(DEFUN SAVESTACK ()
   (LIST GCCOUNT BEGA ENDA P))

(DEFUN RESTORESTACK (DESCRIPTION)
   (COND ((= GCCOUNT (CAR DESCRIPTION))
          (SETQ BEGA (CADR DESCRIPTION))
```

```
        (SETQ ENDA (CADDR DESCRIPTION))
        (SETQ P (CADDDR DESCRIPTION)))
       (T (SETQ BEGA 0)
          (SETQ ENDA 0)
          (SETQ P 0))))
```

How are these used? In forming a saved control state, one would normally call expose and forget the part of the stack that is in use. Instead, we just put into the saved control state object the value of L and a stack description returned by SAVESTACK. When we resume the saved state, we must pass the description to RESTORESTACK before restoring the value of L.

To be truly safe, we must make sure that the user never gets the value of L out of the saved control state without our knowing it. If he did so, he could hang on to it while resuming the saved state, returning up the stack, and then reusing locations which are part of the old L that he is looking at. To do this, we need only make the saved control state a slightly funny Lisp object: whenever the user tries to look inside it, the stack description is changed to all zeros. The user is welcome to look at the value of L inside the saved state, but if he does so the system automatically abandons its plans to keep using the saved stack as a stack.

Micrprogrammed Architectures

Phantom stacks are especially attractive as part of stack-based machine architectures. Primitives for phantom stack operations, implemented in hardware or microcode, will often be just as fast as ordinary stack operations would be, and require only a small increase in the complexity of the hardware or the microcode because of their simplicity and modularity. For this reason, phantom stacks are likely to be used by the microprogrammed interpreter in the next version of the Scheme microprocessor chip [SChip].

However, even when code must be compiled to run on a machine which has only ordinary stacks, phantom stacks still offer an advantage over simply allocating in the heap and garbage collecting.

Upward Funargs and Downward Funargs

In thinking about the use of FUNCTION, a distinction is often made between "downward" use, in which the closure or *funarg* produced by FUNCTION is passed down to inner levels of recursion only, and "upward" use, in which the closure is returned and continues to exist after the stack frame which it points to would normally have been popped. Downward funargs can be implemented safely with an ordinary stack. Indeed, many systems which are based on

ordinary stacks implement funargs that work only downward but leave it up to the user to make sure he doesn't try to return them upward.

Since a downward funarg can be implemented properly with an ordinary stack, one might hope that when funargs are implemented using a phantom stack it should not be necessary to dispel any of the stack as long as the funarg is only passed downward. Alan Bawden and Howard Cannon have pointed out how this can be done.

The key is to redefine what it means to "hand out" a pointer to the list of current variable bindings. The natural definition is that a pointer is handed out whenever it is incorporated into a closure by FUNCTION. This causes any use of FUNCTION to dispel the stack. If instead we identify all the actions that go on in normal downward use of a funarg, and then stipulate that those actions don't constitute handing out the pointer, then downward funargs will not dispel any stack. But to make upward funargs continue to work, we have to be careful to detect the upward activity soon enough to dispel the stack before it is lost.

The actions involved in downward use of funargs are creation of the funarg with FUNCTION and passing of the funarg as an argument in a function call. We can assume that this pushes the argument onto a stack of function calls. And as long as these are the only operations performed on the funarg, we know it is safe not to dispel the stack of variable bindings. So if we make all other operations on funargs dispel the stack if necessary, we will achieve the goal. This means that we may need to call DISPEL whenever

the user looks inside the funarg to find the environment pointer.

the pointer to the funarg is stored anywhere except as part of passing an argument (because it might be accessible there after the stack frame it points to is gone).

the funarg is returned as a value from a function (because it might be returned up past the level at which it was made).

Think what this implies: any time a value is stored in memory, or returned from a function, the system must check whether it is a funarg, and if so must dispel stack as necessary. This includes the functions RPLACA, SETQ and CONS. So we no longer have such a simple, modular scheme which only one part of the system has to know about. Still, for this particular application it might be worth while.

Care is also necessary when a funarg is applied. When we apply a funarg we must temporarily set L from the value saved in the funarg. When we return from the funarg, we must restore L. If we are not careful, inside the funarg where L is set to a pointer to the middle of the stack, we could get confused and

start pushing and popping there, clobbering other parts of the stack. Actually, we want pushing and popping done inside the application of the funarg to be done at the end of the stack rather than at the place where the saved environment points. Here is how we can do that:

```
(DEFUN BINDL (NEWL)
  (LET ((OLDL L))
     (SETQ L NEWL)
     (PUSH NIL)
     OLDL))
```

BINDL sets up a new value of L, temporarily, and returns the old value of L to be saved somewhere (on a function call stack) to be handed to UNBINDL when the funarg returns:

```
(DEFUN UNBINDL (OLDL)
   (SETQ L OLDL)
   (AND (< L ENDA) (>= L BEGA)
        (SETQ P L)))
```

When we apply the funarg, we set L to the saved environment pointer and immediately push something which will not make any difference (NIL in an alist merely reiterates that the value of NIL is NIL). This way, all pushing and popping inside the execution of the funarg will never pop all the way down to the pointer that came from the funarg. It will only get as far as the location containing the dummy entry that we pushed, and this is at the end of the stack.

Yes, this is a kludge. For most stacks it would not be worthwhile, but it might be desirable in this specific application.

How Much Is To Be Gained

Let's attempt to estimate the amount of speed which could be gained by the use of phantom stacks in the Scheme microprocessor chip.

It is easy to construct examples in which phantom stacks save all of the conses, or none of them. Since the savings are so dependent on the details of the code being executed, it is not very useful to measure any particular examples of code.

However, most of the consing done in the scheme chip is done to construct the stacks used by the interpreter, and in most situations all of this consing will be saved. Only when internal lambdas (closures) or catch tags (saved control states) are used do we fail to save all of it.

Since the Scheme microprocessor spends 80% of its time garbage collecting, we can expect saving all the conses to save 80% of the time used. So it reasonable to expect that savings of a factor of two or three in total execution time will be commonplace.

Comparing phantom stacks with other methods of almost-stacklike allocation is difficult, because both will probably save most of the consing in typical applications. Another method may save a few more conses, but will probably make the actual pushing and popping slower. Which one wins depends critically on the code being executed.

Acknowledgements

Alan Bawden and Howard Cannon, inspired by an early version, contributed an idea which is now included.

Bibliography

[BW]

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

[Dream]

Guy L. Steele, and Gerald J. Sussman, The Dream of a Lifetime: A Lazy Scoping Mechanism. MIT AI Memo 527.

[FUNCTION]

Joel Moses, The Function of FUNCTION in Lisp. MIT AI Memo 199.

[LM]

Daniel L. Weinreb and Dave Moon, Lisp Machine Manual. MIT AI Lab, January 1979.

[Scheme]

Guy L. Steele, and Gerald J. Sussman, The Revised Report on SCHEME, a Dialect of Lisp. MIT AI Memo 452.

[SChip]

Jack Holloway, Guy L. Steele, Gerald J. Sussman and Alan Bell, The Scheme-79 Chip. MIT AI memo 559.