ACM Sigplan Notices **27**,3 (March 1992),24-34.

CONS Should not CONS its Arguments, or, a Lazy Alloc is a Smart Alloc

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436, (818) 986-1436 (818) 986-1360 (FAX)

November, 1988; revised April and December, 1990, November, 1991.

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

Abstract

Lazy allocation is a model for allocating objects on the execution stack of a high-level language which does not create dangling references. Our model provides safe transportation into the heap for objects that may survive the deallocation of the surrounding stack frame. Space for objects that do not survive the deallocation of the surrounding stack frame is reclaimed without additional effort when the stack is popped. Lazy allocation thus performs a first-level garbage collection, and if the language supports garbage collection of the heap, then our model can reduce the amortized cost of allocation in such a heap by filtering out the short-lived objects that can be more efficiently managed in LIFO order. A run-time mechanism called *result expectation* further filters out unneeded results from functions called only for their effects. In a shared-memory multi-processor environment, this filtering reduces contention for the allocation and management of global memory.

Our model performs simple local operations, and is therefore suitable for an interpreter or a hardware implementation. Its overheads for functional data are associated only with *assignments*, making lazy allocation attractive for *mostly functional* programming styles. Many existing stack allocation optimizations can be seen as instances of this generic model, in which some portion of these local operations have been optimized away through static analysis techniques.

Important applications of our model include the efficient allocation of temporary data structures that are passed as arguments to anonymous procedures which may or may not use these data structures in a stack-like fashion. The most important of these objects are functional arguments (*funargs*), which require some run-time allocation to preserve the local environment. Since a funarg is sometimes returned as a first-class value, its lifetime can survive the stack frame in which it was created. Arguments which are evaluated in a lazy fashion (Scheme *delays* or "suspensions") are similarly handled. Variable-length argument "lists" themselves can be allocated in this fashion, allowing these objects to become "first-class". Finally, lazy allocation correctly handles the allocation of a Scheme control stack, allowing Scheme continuations to become first-class values.

1. Introduction.

Stack allocation of objects in higher level programming languages is desired because it is elegant, efficient, and can handle the great majority of short-lived object allocations. Traditional higher-level languages such as Algol, Pascal and Ada have preferred to perform all *automatic* storage management by using a stack, while non-stack allocation remains the responsibility of the programmer. However, the limitations of stack allocation are semantically confining, because a strict last-in, first-out, (LIFO) allocation/deallocation ordering does not allow for important classes of program behavior such as that of returning a functional argument as a result. Therefore, the programmer is forced to use complex and error-prone techniques to simulate this behavior himself, even though the abstract programming language may be capable of expressing the behavior more elegantly and directly using, for example, functional arguments as results. Modern higher level languages such as Lisp, Smalltalk, Mesa, Modula-3, ML, and Eiffel, seek to escape these LIFO restrictions to gain in expressive power while retaining elegance and simplicity in the language. Insofar as they succeed, they can greatly improve engineering productivity and software quality.

A cost must be paid for this flexibility through the increased use of heap allocation for objects in the language. Yet the vast majority of objects obey a straight-forward last-in, first-out (LIFO) allocation semantics, and could profitably utilize stack allocation. One would therefore like to provide stack allocation for these objects, while reserving the more expensive heap allocation for those objects that require it. In other words, one would like an implementation which retains the efficiency of stack allocation for most objects, and therefore does not saddle every program with the cost for the increased flexibility—whether the program uses that flexibility or not.

The problem in providing such an implementation is in determining which objects obey LIFO allocation semantics and which do not. Much research has been done on determining these objects at compile time using various static methods which are specific to the particular type of object whose allocation is being optimized. Unfortunately, these techniques are limited, complex and expensive. We present a technique which acts more like a *cache* or a *virtual memory*, in the sense that no attempt is made to predict usage at compile time, but the usage is determined at run time. In other words, the system learns about the usage of objects "on-the-fly".

The major contribution of this paper is the recognition that a wide variety of stack-allocation optimizations are all instances of the same underlying mechanism—*lazy allocation*. Using this insight, we can simplify hardware architecture and language implementations by factoring the problem into two abstraction layers—language implementation using generic storage allocation, and the implementation of generic storage allocation using lazy allocation. Safety is also enhanced by the elimination of "dangling references" due to objects escaping a stack-allocated scope. While lazy allocation already provides for stack-allocation of arguments and temporaries, a programmer can extract even better performance by utilizing "continuation-passing style" to stack-allocate function results.

2. Stack Allocation is an Implementation Issue, not a Language Issue

The stack allocation of variables and contexts in higher-level compiled languages such as C, Pascal and Ada has been a fact of life for so many years since its introduction in Algol-60 that most programmers today assume that stack allocation is a *language*, rather than an *implementation*, issue. Stack allocation cannot be a language issue, however, since the most direct mapping of the nested lexical variable scopes in these languages is a *tree*, not a stack. Rather, stack allocation was a conscious decision on the part of these language designers to provide the *implementation efficiency* of a stack as a storage allocation mechanism even though this choice substantially compromised language elegance. We have also since learned that stack allocation of variables and contexts severely limits the expressiveness of a programming language. Indeed, many of the advances in programming languages after Algol-60 can be characterized as attempts to ameliorate the restrictions of stack allocation. For example, most of the functionality of Simula-67 could have been achieved in Algol-60 through the dropping of the stack allocation requirement along with the syntactic restrictions on procedure arguments and returned values.

The stack allocation of Algol-60 provided a major advance over Fortran's static allocation. Functions could be arbitrarily nested, and recursion became possible. So long as the basic values being manipulated were numbers and individual characters, stack allocation proved remarkably expressive. With the advent of dynamic strings of characters and larger dynamic objects such as arrays, stack allocation started to break down. PL/I used pointers and explicit allocation/deallocation to avoid the limits of stack allocation. Substantial arguments in the 1970's raged about the allocation of function-calling and variable-allocation contexts—"retention" (non-stack allocation) versus "deletion" (normal stack allocation) [Berry71] [Fischer72]. Non-stack-allocation has become steadily more important as the sophistication and complexity of programs have increased. For example, in modern "object-oriented" programming style, most objects are heap-allocated rather than stack-allocated.

Unfortunately, heap-allocation is substantially less efficient than stack-allocation. As a result, programmers constantly seek to utilize stack allocation whenever possible. This change requires much work, because the source code changes required to change from one sort of allocation to another are substantial, even when the logic of the program has not changed. Most importantly, the programmer is required to use heap-allocation for entire classes of objects, even when the vast majority of these objects can be safely stack-allocated. This results from the difficulty in determining at design or compile time *which* of the objects can be safely stack-allocated, because it depends on a particular pattern of function calls, which in turn depends on the input data.

The use of different constructs and mechanisms for heap allocated objects than for stack-allocated objects is therefore less productive for the programmer and less efficient at run-time. It is also an invitation to disaster, because using stack allocation inappropriately can cause a program to fail in a spectacular manner.

We believe that the intertwining of an implementation mechanism (storage allocation) with a programming language mechanism (variable and function scoping) is confusing to the programmer and inefficient for the hardware. It is a violation of the design principle of providing "levels of abstraction", wherein each level can be understood in its own terms, and not require a detailed understanding of every lower level. We therefore describe a mechanism called "lazy allocation" which can be used to provide a uniform interface to the programmer, while taking advantage of stack-allocation as much as possible.

The simplification in language design and implementation permitted by lazy allocation through improved abstraction is important even if additional run-time efficiency on current hardware is not immediately forthcoming. The problem factorization by lazy allocation should allow for more efficient hardware architectures as well as language implementations, which will eventually translate into improved price/performance ratios.

3. The Model.

The standard model of a run-time system in a modern high level language consists of a traditional execution *stack* formatted into stack *frames*, each of which represents the execution state of a particular invocation of a particular procedure. The model may also includes a *heap*, which contains objects that cannot be allocated and deallocated in a last-in, first-out (LIFO) manner. Typical languages using this model are Pascal, C and Ada (without tasks).

C and PL/I make most storage allocation and pointer management the responsibility of the programmer. In these languages, he is allowed to take the address of an object on the stack and store this pointer into an arbitrary variable. A common instance of creating references to local variables is when local variables are passed as arguments to another procedure *by reference*. Passing large objects by reference is usually cheaper than passing them by value, but by-reference argument passing can lead to a dangling reference if a reference into the stack is then stored into a variable having a larger scope, such as a global variable. A dangling reference can cause bugs if it is dereferenced outside of the stack object's lifetime. Thus, the power to take the address of an object on the stack can lead to efficient programs in which temporaries are stack-allocated, but this power can also produce bugs which are difficult to find.

Our model detects these objects whose references are about to escape from the lifetime of their enclosing stack frame, and relocates these objects into the permanent heap. This graduation from temporary to permanent status we call *eviction*. Due to the constant threat of sudden eviction, any access to such an object must be capable of detecting this movement so it can find the relocated object at its new address by following a forwarding pointer.[1] The mechanisms for dealing with objects which can be unexpectedly moved are well known in the context of "on-the-fly", or "incremental" compacting garbage collectors [Baker78][Lieberman83].

Once we have installed the machinery necessary to deal with objects which can suddenly move, we can contemplate the possibility of allocating *everything* initially on the stack. If we implement an "escape alarm system" to detect escaping references, we can use these alarms to trap to an eviction routine which will transport evicted objects into the heap. We call the policy of stack allocation followed by eviction traps "lazy allocation", because the system is too lazy to perform the work involved in heap allocation until this work is forced by the object's usage.

In the lazy allocation model, we have a global heap, a stack formatted into stack frames, and a small number of machine registers. We posit the existence of an ordering relation on object addresses which adheres to the following axioms:

1. The location address of an object in the global heap compares lower than the location address of an object in the stack.
2. The location address of an object in a frame near the stack top compares greater than the location address of an object in a frame near the base of the stack.

Lazy allocation will enforce the following rule:

Temporary Restraining Order — No temporary object in the stack can be pointed to by an object whose location address compares lower; i.e., no object in the heap can point into the stack, and no object in the stack can point "up" the stack, only "down" the stack.

The Temporary Restraining Order (TRO) is enforced by checking every primitive operation which could possibly violate it. The only operation which can violate TRO is the storage of a pointer into a stack frame (or the heap) which compares lower than the object pointed at. In other words, when performing an assignment, we need only check that the ordering is preserved:

```
component(p) := q;        /* p,q pointers; assert(p≥q). */
```

Statistically, most component assignments in higher level languages will respect the TRO rule, because component assignments are usually used to initialize components of newer data structures to reference older ones.[2] The main exceptions to this observation are assigning to (more) global variables and returning values. By assigning a pointer to a global variable, we are very likely to violate TRO because our lazy allocation will allocate everything on the stack.

Since most function returns result in binding a variable in the caller's frame, they can have the same effect on returned values as an assignment to a global variable—eviction. In some cases, this eviction can be avoided by having the caller allocate the space for the returned result in his own frame, and passing a reference to this space as an additional argument. This technique, which we call "caller result allocation" has been used since the early 1960's in Fortran, Cobol and PL/I compilers. There are two problems with caller result allocation. First, when the called routine attempts to initialize any components of this structure, these assignments may cause eviction of the components. Second, caller result allocation only works when the caller knows the size of the object being returned in advance. Nevertheless, it can be used to reduce the number of evictions of returned values.[3]

When a TRO violation is about to take place, the system executes a "transporter trap" [Moon84], which evicts the target object of the pointer from the stack into the heap. The process of relocating this object can cause recursive transporter traps, however, when pointers which are components of the object being relocated are discovered to also point into the heap. If relocation of the object, installation of forwarding addresses, and updating pointer components are all performed in the correct order, this recursion will terminate. When the recursion terminates, not only has the object causing the first trap been relocated out of the stack, but so have all of the objects accessible from it. When the original trap has completed, the updated address is then used to complete the assignment operation which caused that trap.

Since forwarding addresses are left for every non-functional object which is evicted in this manner, references from machine registers and from objects still in the stack to the evictee can follow these forwarding addresses to find the moved object. A functional (read-only) object still in the stack does not need a forwarding address because the stack original is equivalent to the heap copy,[4] and continues to function correctly until the frame is exited, at which point it is abandoned because no live references to it exist. Thus, the transporter traps caused by attempted violations of the TRO rule serve to relocate the objects so that the TRO rule is preserved, and so that the semantics of the objects themselves are also preserved.

---

[1] Forwarding pointers are only required for side-effectable objects; functional (read-only) objects are copied, but do not require the detection or following of forwarding pointers [Baker93].

[2] On architectures without special TRO-checking hardware, common cases like initialization can be easily recognized and optimized.

[3] Functions in expression-oriented languages like Lisp will often return values which are never used when the function is called for its side-effects rather than for its returned values—e.g., `print`. An important optimization when using lazy allocation is to inform a called function when results are not expected, so that eviction of unneeded results is not performed; this *result expectation* optimization is discussed in a later section.

[4] If the language offers an address-comparison operator (e.g. Lisp's `EQ`) for functional objects, then forwarding pointers will be required. [Baker93] argues for a more comprehensive and portable treatment of such an *object identity* predicate so that functional objects can be relocated without requiring the checking or leaving of forwarding pointers.

Let us call a stack which does not violate TRO *well-ordered*. By preserving the Temporary Restraining Order, we have the following theorem:

Theorem. If a frame from a well-ordered stack is popped, and the registers contain no references to the popped frame, then no dangling references are created by the pop.

Proof by induction. The only dangling references could come from the registers, further down in the stack, or the heap. By hypothesis, the registers have no pointers to the popped frame. There is no higher stack frame, hence no references from the stack above. Due to transporter traps, there are no references from down the stack or from the heap. Therefore, the only references to the popped frame are from the popped frame itself, in which case the entire frame is garbage. QED.

Since the stack in our model is restricted to *transient* objects, it is like a "flophouse" hotel. Since such an address is not respectable, one does not give it out. In order to put down "roots", one must move to a more respectable address in the permanent heap.

4. Complexity

Stack allocation has recently come under attack as being slower than garbage-collected heap allocation under certain circumstances [Appel87a]. Since many of the benefits of lazy allocation are lost in this instance, we must first demonstrate that stack allocation retains its advantage under most conditions.

As [Baker78a] and [Appel87a] demonstrate, the cost of heap allocation can be reduced to approximately the same as stack allocation. This reduction is achieved by increasing real memory size (relative to the program) so that the number of garbage collections (which are really heap copies) is reduced. With enough memory, the amortized cost of garbage collection (heap copying) approaches zero, so that the cost of allocation (incrementing a pointer) becomes the dominant cost. Since the cost of stack allocation/deallocation is also the incrementation/decrementation of a pointer, the costs of heap and stack allocation become similar.

This reduction in execution time is only achieved, however, through a tremendous increase in the size of real memory. Using the model in either [Baker78a] or [Appel87a], the allocation time becomes independent of the memory size only when the fraction of space used is negligible—i.e., at least an order of magnitude smaller! Even with the exponential reduction in memory prices over the last 30 years, CPU prices have fallen even faster, resulting in an ever-increasing fraction of system costs tied up in memory. These trends indicate that the use of garbage collection to simulate stack allocation is a waste of resources. Stack allocation, on the other hand, operates with the same efficiency (for those objects having LIFO behavior) whether memory is nearly empty or nearly full. Therefore, the "space-time product" for stack allocation is better than that for garbage collection whenever the utilization of memory is greater than a small fraction.

The growing popularity of "generational garbage collectors" [Lieberman83] [Unger84] [Appel89] is an indication that the efficient use of real memory is relatively important in most applications. Lazy allocation can be viewed as a variation on generational garbage collection in which individual objects are generations, and tenuring to the heap is immediate.

Because lazy allocation copies objects, it is an interesting exercise to compare the complexity of our model with that of a straight-forward copying garbage collector such as Cheney's [Cheney70]. We have shown in [Baker78a] that the amortized cost of allocation in a system with a copying garbage collector is linearly proportional to the amount of space being allocated; the constant of proportionality is a more complex expression depending upon the occupancy factor of the total amount of storage under management. Since we must usually initialize an object being allocated, and this initialization process is usually linearly proportional to the size of the object being allocated, the most we can hope for from our lazy stack model is a better constant of proportionality than in the uniform heap-allocation model.

Allocating an object on the stack is virtually the same process as allocating an object in a compact heap because it involves only a movement of the free space pointer. Deallocation of a stack frame is essentially free, since the stack frame need not be scanned. Well-ordering violations can occur, however, resulting in the relocation of objects from the stack. The total size of the relocated objects during a recursive transportation trap cannot exceed the current size of the stack, and once an object is relocated to the heap, it will not be relocated again.[5]

Thus, in the worst case, every object is allocated on the stack, and is then relocated to the heap. This is a worst case in storage, because since every object was allocated once on the stack and then relocated to the heap, the storage on the stack is no longer occupied (until the frame is popped). It is a worst case in time, because every object is allocated first on the stack and then relocated to the heap, when it could have been allocated directly on the heap in the first place. If the cost of moving an object exceeds the cost of building it directly on the heap, then our model will result in poorer performance than the pure compact heap model. (This analysis does not count the added costs of constantly checking for objects which may have moved, so that their forwarding pointers may be followed.)

The literature indicates, however, that while the worst case *may* occur, the most common cases are LIFO allocation of objects. If this is the case, then most objects will get allocated on the stack, and will become inaccessible before the stack frame is popped.[6] As a result, most of these objects will never be evicted from the heap. Those objects which are evicted will require somewhat more effort to get them relocated to the heap, but in these cases we do not begrudge the additional effort because 1) these objects are still accessible, and 2) we have saved so much time as a result of stack allocating the vast majority of objects, that we can afford to be more generous in these few instances.

The programmer himself usually has a very good idea about which allocated objects are likely to have stack-like extents, and which objects are not. If he knows that a particular object is almost certain to cause a well-ordering trap, he can allocate it directly on the heap himself, and save the system the time and expense of figuring this out for itself. The programmer can indicate this information either in the form of a declarative "hint", or by calling an allocation routine with a different name. In either case, the percentage of objects which can be successfully reclaimed by lazy allocation will be greatly increased.

By using "continuation-passing style"—discussed later—the programmer can extend the lifetime on the stack of values returned from functions, and thereby gain efficiency not easily obtained through more traditional optimizations. We conjecture that the majority of objects reclaimed early by "ephemeral" or "generational" garbage collectors are extremely temporary objects which could be more efficiently stack-allocated using continuation-passing style and lazy allocation.

Lazy allocation utilizes forwarding pointers to correctly preserve "object identity". Since we have assumed that the heap already utilizes forwarding pointers in its management, lazy stack allocation will exact no additional penalty. Brooks's forwarding scheme [Brooks84] can be used to advantage, since its overhead on a cached architecture is rather small. As we discuss later, functional objects require no forwarding pointers, in which case all overhead is concentrated into assignment operations.

In real architectures, there may be additional savings from our model that will not show up in raw instruction counts. Modern architectures have smaller, faster memory banks which *cache* the most heavily used memory locations. Between the effects of caching and paging, the amortized cost of cached memory references can be up to *two* orders of magnitude faster than the cost of non-cached memory references [Jouppi90]. Since our model attempts to keep everything on the stack as long as possible, it is likely to have more local behavior than a model which allocates objects directly on the heap.[7] If the hardware memory system knows that this is a stack, additional optimizations can be made, such as not writing back popped stack frames to the backing store.

Another optimization possible with our model is the inlining of the allocation routine itself. In this case, the initializing information is copied directly into the appropriate locations. For example, a Lisp-like z:=CONS(x,y) is just "t1:=x;t2:=y;z:=addr(t1)", where t1,t2 are

[5] We here assume that there is no sharing of functional objects, since eviction unshares them due to the lack of forwarding pointers. This assumption is generally true, but if functional objects are to be highly shared, then forwarding pointers should be checked and left as if they were non-functional objects.

[6] These statistics can be improved through the *result expectation* optimization, discussed later.

[7] [Stanley87] reports a phenomenally large reference locality for stack caches, as opposed to other data references.

adjacent locations *known to the compiler*, and the compiler can therefore optimize out the incrementation of the allocation pointer. In a system without a lazy allocation routine, one would need additional register shuffling, followed by storage to an unknown location of memory, even when the allocation routine itself was inlined. Such a routine would be slower than our lazy allocation—perhaps as slow as an out-of-line procedure.

A new generation of shared-memory "symmetric multiprocessing" (SMP) machines is starting to be utilized for tasks requiring flexibly allocated storage. On such machines a global storage allocation routine must be protected by a lock to avoid inconsistency and this lock can become a performance bottleneck. If the allocation responsibility is split up so that each processor has its own allocator, this bottleneck is removed. Lazy allocation elegantly achieves such a split, because each processor already has its own stack, and lazy allocation may reduce the allocation demand enough so that permanent allocations can be made using a global, shared allocator without undue contention.

5. Applications

A. Malloc/New

The most straight-forward application of lazy allocation is a version of C's `malloc` or Pascal/Ada's `new` which allocates the object within the current stack frame—expanding it if necessary. In other words, C's `malloc` becomes equivalent to the old `alloca`. With lazy allocation, one never calls the heap `malloc`/`new` directly, but leaves it to the eviction trap to call these routines. Since ephemeral objects will be deallocated automatically when the enclosing stack frame is exited, `free` (`dispose`) needn't do anything when handed a pointer into the stack. However, for a small cost, some error checking can be gained by marking the stack object as free, so that any access or attempted eviction will cause an error trap instead.

B. Functional Arguments

The correct implementation of functional/procedural arguments ("funargs" [Moses70]) in higher level languages is quite complicated. The creators of the DoD standard Ada language were so fearful of these constructs, that they were summarily banned from the language [STEELMAN78, Requirement 5D]. In transmitting a functional/procedural argument, not only must a pointer to the executable instructions be passed, but also a pointer to the environment of the function. This is because a function defined at a lexical level other than the topmost level may refer to local variables of another function. These variables are not local to the function being passed, and they are referred to as "free" variables.

Some languages such as C finesse the free variable problem by not allowing functions to be defined at other than the topmost lexical level. In this case, all free variables are global, and since global variables can be statically allocated, their addresses can be embedded into the instruction stream so that an additional environment pointer need not be passed. Much expressive power is lost in the language by this restriction, however.

More powerful languages such as Algol, PL/I, Pascal and Lisp allow the definition of functions and procedures within inner lexical contexts, and therefore must package up pointers to both the code and the environment when passing a function/procedure as an argument. While this involves more work, these functional/procedural arguments are strictly more powerful than C-style functional arguments. True functional arguments are capable of simulating arbitrary data structures—at least within their lifetimes—and therefore the work necessary to allocate a new structure for their implementation is required.

Algol, PL/I and Pascal have carefully restricted the use of functional arguments to make sure that they can never escape the lifetime of the stack frame in which they were created. As a result of these restrictions, functional arguments can be safely allocated on the stack. Therefore, these objects are not "first-class". Part of the reason for these language restrictions has to do with dangling references. If a functional argument refers to objects on the stack, and the stack is popped to the point where these objects are no longer valid, then a dangling reference will be created which can cause nasty bugs.

The restrictions on the use of functional arguments in Algol, PL/I and Pascal are rather arbitrary and constrain the expressive power of the programming language, although not so much as the restrictions of the C language. Several modern languages like Common Lisp and Scheme offer "first-class" functional arguments, which can be returned from functions and stored in data structures. However, the efficiency impact of this freedom is normally quite severe. Not only must the functional arguments themselves now be allocated on the heap, but much of the normal variable-binding environment must also be heap-allocated. Only in this way can dangling references be avoided.

Lazy allocation can solve the problem of allowing functional arguments to be first-class, while keeping stack allocation for the vast majority of these objects that do not escape the lifetime of their creators. In the case of Lisp languages like Common Lisp and Scheme, the environment stack is kept separate from the control stack, so that lazy allocation "almost" works without any change. A problem which occurs in any language which offers both functional arguments and side-effects, however, is making sure that the object being side-effected is "the" object, rather than a copy of it. In particular, when one creates multiple funargs which share the same free variable, and one or both of the funargs side-effect this variable, it is important that the side-effect be visible to all of the funargs. The usual solution to this problem is to allocate a unique assignable "cell" in the heap for each such variable, which is then bound as the "value" of the variable; [Kranz88] calls this transformation *assignment conversion*.[8] When multiple funargs are created which reference this variable, then each will then reference the same cell. The Common Lisp example below exhibits this situation.

```
(let* ((x 3))          ; Create cell x initialized to 3.
  ((lambda (y z)
     (let* ((ex x)     ; Get the current value of x (= 3).
            (ey (y))   ; x <- 4
            (ez (z))) ; New current value of x (= 4).
       (list ex ey ez)))
   #'(lambda () (setq x 4) x)  ; Funarg with free var. x.
   #'(lambda () x)))  ; Funarg with free var. x.
=> (3 4 4)
```

Unfortunately, assignment conversion is very expensive, because it causes *every* variable to be bound to such a heap-allocated cell, unless it can be statically shown that either 1) the cell is never assigned to after initialization, or 2) the cell is never shared among funargs [Kranz86] [Kranz88]. Using lazy allocation, however, we can cheaply allocate an assignable cell for every variable in the same stack frame as the variable itself, and this cell will only be relocated to the stack when it tries to escape from that stack frame's lifetime.[9] Thus, lazy allocation can be used for both the funarg environment itself, as well as the assignable cells, so that in the most common situations all of the allocations for funargs occur on the stack and are never relocated to the heap.

In the case of non-Lisp languages, where the environment stack and the control stack are usually merged, we must be more careful. The simplest solution would be to utilize lazy allocation on the stack frames themselves, which would work because any frame relocated into the heap would leave a forwarding address for any other object which pointed to it. Unfortunately, the eviction of one stack frame has the undesired effect of immediately evicting all of the stack frames nearer to the bottom of the stack as well. When using such a policy, the top stack frame's eviction causes the entire stack to be relocated into the heap. While there may be occasions where this massive relocation is desired (see the later section on Scheme continuations), this policy is usually overkill for funargs.

A better solution for implementing first-class funargs in non-Lisp languages would be to more closely follow the Lisp example. A funarg environment need only retain the bindings of the variables free in the function being passed, so the funarg environment could be a simple vector of these variable values. Of course, the same "assignable cell" indirection must be made for free variables in non-Lisp languages that we demonstrated above for Lisp. Since lazy allocation allows both the assignable cells and these environment vectors to be (initially) stack-allocated, we get excellent performance until a funarg is returned or stored into a global variable and becomes first-class.

C. Lazy Argument Evaluation

Lazy evaluation of the arguments of a function call is the deferral of the evaluation of the argument expression until the argument is actually "used". Lazy evaluation of arguments can be more expressive and efficient than so-called "applicative" evaluation if an

---

[8] Erik Sandewall circulated a memo describing the same technique in 1974 [Sandewall74]; Greenblatt utilized this idea in the MIT Lisp Machine [Greenblatt74].

[9] On a machine with a cache, the variable and its cell should be initially located in the same cache line, so that the additional indirection through the variable reference to the cell will execute as quickly as possible.

argument is never used, because the argument expression will never be evaluated.

Lazy evaluation appeared in Algol-60 under the guise of "call-by-name", and was implemented by means of "thunks" [Randell64]. Thunks are quite similar to functional arguments, in that they have some executable code and an environment in which to execute the code. In fact, the semantics of Scheme's `delay` expressions are given in terms of functional arguments [IEEE-Scheme90].

As a result of the simple implementation of lazy argument evaluation using functional arguments, lazy allocation will work for lazy evaluation in the same way that it works for functional arguments. Most "lazy" arguments will be evaluated before their defining stack frame is exited, and those that have not, will most likely never be evaluated. However, those that do escape from their defining stack frame will be evicted to the heap, where they will evaluate correctly if the need arises.

The possibility of escaping Scheme `delays` suggests an optimization that may sometimes be beneficial. If a `delay` is about to escape, one may want to arrange for its immediate ("strict") evaluation. If the `delay` is functional, then the time of its evaluation cannot affect its value, yet this value may occupy less space than the `delay` itself (zero if the value is an "immediate" quantity such as a floating point number), thus improving heap utilization and performance. We call the evaluation order resulting from this optimization "downward lazy evaluation", due to its similarity to "downward funargs"—i.e., those functional arguments that obey stack ordering.

D. Argument "Lists"

Programming language designers have long been tantalized by the similarities between parameter lists and structured values ("records" or "structures"). For example, Ada utilizes nearly the same syntax for declarations of both, and nearly the same syntax for function calls and record "aggregates" [Ada83]. Lisp semantics presume the existence of a Lisp list of arguments [McCarthy65]. Yet nearly all language designers are forced to back down from unifying these two concepts due to the unacceptable loss of efficiency that would result. Making parameter lists into true first-class objects would force them to be heap-allocated, with the concomitant problems of determining when and how to deallocate them.

Lazy allocation offers the language designer the ability to unify record structures and parameter lists without the loss of efficiency. Every function and procedure can be elegantly defined as accepting just one parameter—a record structure. A function/procedure call constructs an argument object on the stack by creating a new instance of this structure initialized with the individual argument components. A pointer to this argument object is then passed to the function or procedure, where a "destructuring pattern match" of the argument is made using the parameter record definition as a pattern. Elegance and power are obtained in two ways: the capabilities of record structures are made available to argument objects, and the capabilities of argument objects (e.g., pattern-matched *destructuring*) are made available to record objects.[10] Even more fascinating is the transfer of other argument-passing ideas to data structures—e.g., lazy evaluation becomes lazy component initialization [Friedman76].

In the most common case, argument objects are immediately destructured, and do not escape the lifetime of the called function or procedure, and therefore can be deallocated when the function/procedure returns. However, should the argument object become first-class, it is evicted from the stack, and is thereby retained when the function/procedure returns.

By giving argument objects a (potentially) first-class existence, some efficiencies can be obtained. For example, some recursive procedures pass on a number of arguments unchanged to successive recursions, where they are used only when the recursion terminates. By passing a pointer to a portion of the argument object instead of copying these arguments, some effort can be saved. Furthermore, if the number and/or size of these arguments are related to the depth of the recursion, an algorithm that is quadratic in complexity may be reduced to linear complexity [Dybvig88].

E. ANSI-C STDARG's

The programming language BCPL [Richards74], from which the C language descends, passed all arguments uniformly as a vector on the stack which could be accessed in exactly the same way as any other vector. This clean model depended upon a uniform size for all objects, and was therefore abandoned in C. Until the development of UNIX `varargs` and then ANSI `stdargs`, there was no portable way to pass an arbitrary number of arguments to functions, most especially variants of `printf`. These portable forms offer a kind of "stream" access to the argument "list", and the receiver of the variable argument "list" sequentially reads this stream, and provides the type information necessary to decode it. These streams provide only data, and no pointers to these arguments are allowed.

The `stdargs` facility is currently the only facility that provides stack allocation of variable-sized quantities of storage in ANSI-C; ANSI-C is highly tuned to allow only fixed-size stack frames whose size is known by the compiler.[11] With the demise of `alloca`, which allocated variable-sized objects on the stack, ANSI-C's restriction against taking the address of such an argument is particularly obnoxious, because some applications of `alloca` could have been (painfully) simulated using variable-sized `stdargs`.

Both `stdargs` and `varargs` are ugly and non-modular, and introduce notions not used elsewhere in C. Because C wants to charge all costs for variable-length argument lists to those functions which use them, these forms must not interfere with the passing of fixed-length argument lists in registers (the norm on RISC architectures).[12] Because the simplest implementation of a stream is the incrementing of a pointer variable through a memory structure, the first step in most implementations of `va_start` is to store all of the register-passed arguments into memory, and then utilize simple pointer-stepping for all arguments. Indeed, given the fact that the argument-reading stream may be passed on to additional functions, it is difficult to conceive of a compiler smart enough to implement `stdargs`/`varargs` in any other way. The inability to create a pointer to a `stdargs` argument is therefore unreasonably restrictive, since it almost certainly resides in addressible memory. Presumably, the no-pointers restriction is meant to protect the user from the particularities of storage allocation of the storage needed for the variable-length argument list, which may be allocated by `va_start` and deallocated by `va_end`. Such protection is out of character for C, since no such protections exist for other stack and heap-allocated objects. Most implementations allocate the storage needed for `va_start` on the main C stack (using a version of the now-banned `alloca`), however, in which case the `va_end` is extraneous. Allocating storage for `va_start` in any other place is almost certain to run afoul of `longjmp`, which must then decode the stack and execute `va_end` for each stack frame involving `stdargs` for which `va_start` was executed, but `va_end` was not.

A lazy allocation mechanism could dramatically simplify the `stdargs` device of ANSI-C. A new, first-class polymorphic "stream" data structure could be defined which could be opened and read sequentially from a variable-length argument list. Since this data structure would exist either in the stack frame storage of either the calling or the called subprogram, no restrictions on taking addresses would be needed. If any pointers survived, the targeted objects would be automatically moved into the heap. Reading such a stream would produce a truncated stream consisting of the rest of the list. If any stream ("`ap`") pointers survived, then the "rest" of the stream (including objects it referred to) would be relocated to the heap.

F. Common Lisp &REST Arguments and Scheme ". z" Arguments

The semantics of the Lisp language were originally defined by a "meta-circular" interpreter which created actual Lisp lists of evaluated arguments as part of its evaluation of function application [McCarthy65]. While most modern Lisp implementations put evaluated arguments onto a stack instead of a list, Common Lisp and Scheme retain one vestige of the original Lisp evaluator—the `&REST` argument. Both Lisps allow for the passing of an arbitrary number of arguments to a function, but the called function must somehow be capable of addressing these arguments. Common Lisp uses normal

[10] So long as these argument objects are *functional* [Baker93], one can still pass argument objects through RISC architecture registers rather than memory, because functional objects can be transparently copied—even to a bank of registers.

[11] ANSI-C doesn't strictly require stack allocation (on the main C stack) of variable-size argument lists, but any other implementation must deal with signals and `setjmp`/`longjmp`, which require main-stack-allocation semantics.

[12] Curiously, arguments passed in registers are required to be stored into memory upon entry to a function, unless the corresponding parameter is declared with a storage class of `register` (which is not the default). An optimization not always performed is to store the argument only if the address-of ("`&`") operator is ever applied to the parameter; this usage can easily be detected by the compiler.

positional matching for the first several arguments, and has a keyword-matching capability, but functions with a large number of relatively homogeneous arguments such as "+" are most elegantly handled using a `&REST` parameter. The semantics of the `&REST` parameter are that it is bound to a Lisp list of the arguments remaining after all required and optional parameters have been bound. Scheme does not have keyword arguments, but does allow the equivalent of Common Lisp's `&REST` parameter which is denoted by putting a symbol in the last "cdr" position of the parameter "list", which is therefore an *improper* list.

Unfortunately, the creation of first-class Lisp lists for handling these `&REST` parameters is quite expensive. Yet to be safe, a true list must be constructed for the `&REST` parameter, since the called function may do anything with this list it likes, including returning it or side-effecting it. For example, Lisp's `LIST` function itself has the trivial definition where it simply returns its `&REST` argument:

`(defun list (&rest args) args)`[13]

Since there are many reasons for passing variable numbers of arguments to a function, and only a few of them involve creating a list, it is unfortunate that Lisp forces a list allocation for this common situation.

The Lisp Machines derived from the MIT Lisp Machine [Greenblatt74] actually do format their argument lists on the stack to look like Lisp lists so that they can be passed as `&REST` arguments and traversed using the normal `CAR` and `CDR` functions. However, these argument lists are not first-class Lisp lists, because the lists so created have the same lifetime as the enclosing stack frame. Therefore, while these `&REST` arguments can be passed down the stack, they can never be stored into the heap, returned past their creation point, or `RPLACD`'ed. However, because they are formatted as lists, they can be passed to other functions as lists—e.g., as the last argument to `APPLY`—and thereby avoid a quadratic explosion of copying [Dybvig88].

In our lazy allocation model, however, *all* CONS'ing is first performed on the stack, so there is no additional penalty for CONS'ing `&REST` arguments. Furthermore, using lazy CONS'ing, `&REST` lists are truly first-class lists, since they are created using exactly the same mechanism that is used to create *any* Lisp list.

(As an aside, we point out that if Lisp argument lists are to be constructed so that the `CDR` pointers always point towards the base of the stack, and if each argument is to be inserted when its list cell is allocated, then this virtually requires that arguments to a Lisp function be evaluated in *reverse* order of appearance.)

G. Tail Recursion

Tail recursion is a Lisp optimization that was elevated in the Scheme dialect into a requirement. By requiring that a tail-recursive routine called to a depth of n is allowed to use only O(1) amount of *control* stack, Scheme can simulate iterative control structures in a storage-efficient manner without a distinct iteration construct. Typical implementations achieve this by reusing the stack frame on a tail-recursive call. The introduction of lazy allocation requires a new understanding of the meaning of a "tail recursion optimization", since any allocation performed during such a loop will increase the size of the stack frame which is being reused, potentially allocating an unbounded amount of stack space. One interpretation is that such a program utilizes additional storage during its execution, and therefore isn't "really" iterative at all. Another interpretation is that the Scheme tail recursion requirement is unreasonable, since a lazy allocation implementation utilizing arbitrary storage space may be more efficient (within its storage limitations) than a more strict stack frame reusing strategy, and the Scheme requirement makes the programmer "subvert" the compiler in order to achieve his wish. A default interpretation is that a tail recursion "optimization" disables lazy allocation, and forces allocation directly into the heap.

H. Scheme Continuations

Scheme is a dialect of Lisp which has a very interesting construct called a *continuation*. A continuation is a functional argument that embodies the "rest of the computation". When a continuation function is called with an argument, it does not act like a normal call, but instead *returns* from a previous expression evaluation. Normally, when a function is called, the arguments are evaluated, an argument list is constructed, the caller is suspended, and control is transferred to the callee. The callee creates a new frame on top of the stack and starts execution. Eventually, the callee *returns* with a *returned value*. A return is usually implemented by saving the returned value in a register, popping the callee's frame from the stack, and continuing execution at the point in the caller's code where the callee was originally called. During the execution of the callee, the suspended caller can be considered a kind of functional argument, which, if called, would execute the "rest of the computation". This "function" even takes an argument—the value to be returned from the callee. This "function"'s main fault is that if it is called, it will never return. We designate this theoretical "function" the *continuation* of the suspended caller program.

In a traditional stack implementation, the continuation has more than a passing resemblance to a functional argument. It consists of a pair of values: the point in the program text where execution will resume, and an environment—the current stack—in which to interpret lexical variable occurrences in the program text. If we could somehow package this continuation into a "real" functional argument, then we could simplify the notion of function calling by always including a continuation argument, and no longer including the return "PC" and the return stack-pointer as integral parts of the function calling sequence. In fact, the "jump to subroutine" operation of most modern computer architectures can be viewed as an optimization of the sequence "push continuation argument; jump to the beginning of the called function".

In the most primitive Scheme model, then, there are no *returns* from subroutines, only *calls* to continuations. The basic difference between a normal function and a continuation is that calling a normal function will *push* onto the stack, while calling a continuation will *pop* from the stack.

Traditional stack implementations of traditional languages work correctly, because under normal conditions all continuations created during the execution of a program have strict LIFO allocation/deallocation behavior. In other words, the continuation (return point, stack pointer) does not escape the lifetime of its creator, and therefore no dangling references are created. In the Scheme language, however, since a function callee can gain access to his continuation through a special construct (`call/cc`), this continuation can escape the lifetime of its creator and become a first-class object. (ANSI-C [ANSI-C88] also defines the operations `setjmp` and `longjmp`, which allow for the "capture" and application of a continuation which is *not* first-class.)

If a system utilizes lazy allocation, then stack frames will remain on the stack, so long as LIFO allocation/deallocation behavior is observed. If a continuation attempts to escape its creator's lifetime, however, its stack frame will be evicted from the stack, which will recursively cause all lower stack frames to also be evicted.[14] An implementation of a language which must deal with the possibility of a stack frame suddenly being moved can be quite inefficient, because virtually every access to the stack frame must check for the existence of a forwarding pointer. *Functional* stack frames—which may still point to assignable local variables—can relax this restriction by allowing copying without forwarding.[15] We can thus obtain a behavior analogous to that of many current implementations of Scheme which copy the entire control stack to the heap when a continuation is captured. Of course, assignable local variables cannot be copied, but must be relocated in order to retain their shared semantics.

We do not contend that lazy allocation solves all problems in the high performance implementation of Scheme continuations, and experience may prove that Scheme continuations are better managed with more specialized techniques. Nevertheless, it is interesting that the generic lazy allocation model faithfully captures the behavior of several existing Scheme implementations, without requiring any special handling.

I. Function Results and the *Result Expectation Optimization*

As we have pointed out above, lazy allocation is not lazy when it comes to result values. This is because results must be returned "up" the stack, usually by assigning them to a temporary value in the caller's frame. This is unfortunate, because the efficient handling of results is as important as the efficient handling of arguments. The efficient allocation of results, however, is a difficult problem.

---

[13] In Scheme, this becomes `(define (list . z) z)`.

[14] Lower stacks frames will be evicted only if they have not already been evicted; this solves the multiple stack copy problem mentioned in [Hieb90].

[15] Stack frames in Scheme may have assignable slots even when no side-effects are performed by the programmer; this behavior is a result of the ability of Scheme continuations to be resumed many different times.

The first optimization for reducing result eviction we call *result expectation*. Since many functions are called "for side-effects" rather than "for result value" in expression-oriented languages like Lisp, the caller should notify the function of this expectation so that unneeded function results can be thrown away instead of evicted.[16] Other functions are called for their result, but only 1 bit of result information is actually used—whether the result matches a distinguished value (`nil` in the case of Lisp, `0` in the case of C). In such cases, we need preserve only the single bit of result information actually needed, to avoid the eviction of large structures which are already mostly garbage.

Another optimization for avoiding the heap-allocation of function results is for the caller to allocate space for the result and pass this space by reference; this technique, which we call "caller result allocation", has been used since at least the early 1960's in Fortran, Cobol and PL/I compilers. Eviction upon callee return is avoided since the callee no longer performs allocation. While this technique is widely used in programming language implementations, it depends upon the ability of the caller to guess the correct size of the result, and it forces the called routine to use side-effects to communicate its result information. When the size cannot be guessed—e.g., in the case of some Ada unconstrained array results—this method fails, and heap-allocation must be used. Heap allocation, however, runs the risk of "storage leakage" in non-garbage-collected language implementations if an error or other non-local transfer of control fails to deallocate this storage when the stack is contracted.

Another method for avoiding the heap-allocation of function results is for the result to be allocated in the stack, but to redefine the caller's stack frame to include this result before returning to the caller.[17] This scheme is similar to caller result allocation, except that the caller conceptually passes the entire "rest-of-the-stack" as the result reference, which is then chopped back to its actual size before being returned. This scheme also works, and has been used in some Ada compilers [Sherman80], but can waste arbitrary amounts of space on the stack if the process is iterated. Consider, for example, a recursive program which allocates a result at the bottom of the recursion. This result, and all the intervening space, will become part of the caller's stack frame, even though most of this space is no longer used. Since one knows the current extent of the stack, one could conceivably copy the result back to "close up" the space, but if this process is iterated, a quadratic explosion of copying could result [Dybvig88]. Therefore, the non-lazy eviction of a result value to the heap just once can be more efficient than trying too hard to keep the result on the stack.

Unlike previous schemes for redefining the stack frame [Sherman80], we suggest that whether the object is relocated to the heap or kept as part of the caller's stack frame should be the choice of the *caller* as part of his result expectation. In other words, the result expectation code to be included in every function call consists of at least the following two bits of information:[18]

ResultCode	Action
00	— don't return a result (used for non-last position of "progn")
01	— nil/non-nil as result (used for boolean position of "if")
10	— evict result if necessary (normal lazy allocation operation)
11	— don't pop frame (redefine caller's frame to include result)

The result expectation code inherited by a function from its caller is used only when the function attempts to return a result (in Lisp, when it executes a function in the "tail-call" position); the rest of the time, it computes its own result expectation code (perhaps dependant upon the caller's expectation code) when calling out to other functions. By propagating result expectation codes, the result consumer can inform the result producer of its wishes regarding the allocation of this result. A programmer or a compiler can therefore use result expectations to avoid evictions when the amount of wasted space is calculated to be within acceptable limits.[19]

J. Extending Result Lifetimes and Multiple Return Values

There is another method for allocating function results on the stack which will not cause immediate eviction. This method depends upon a source-to-source conversion of a program called "continuation-passing-style conversion" (CPS conversion). In converting to CPS, a program is turned inside out so that returns and returned values, which can be viewed as implicit calls to continuations, are converted into explicit calls on explicit continuations with the values as arguments. While the CPS form of a program will execute and produce the same answer as the original program, its execution on a traditional stack implementation will use considerably more stack space. Since there are no longer any returns, neither are there any stack pops, at least until the very end of the program, and so the amount of stack used can be substantial.

The conversion to CPS form has certain benefits, however. Since the stack is retained until the very end of the program, there is never any possibility of dangling references for stack-allocated objects. Therefore, the CPS form of the program can execute correctly even when the original form of the program would have failed due to some stack-allocated object leaving the scope of its creator. In other words, CPS style offers a "retention" rather than a "deletion" strategy for stack frames [Fischer72].

The "continuation-passing style" of programming thus offers new flexibility to the programmer who wishes to utilize stack-allocation whenever possible. If he calls a function with a stack-allocated argument which could then become part of that function's returned value, he is likely to get a dangling reference without lazy allocation, or cause the eviction of a large structure with lazy allocation. However, he can postpone the eviction for a while by calling that function with an explicit continuation which will accept the "returned" value and continue executing without popping the stack or causing any evictions.

The most trivial example of all is the Lisp `CONS` function itself which allocates a list cell. If implemented as a true Lisp function in a system using lazy allocation, the list cell would be allocated on the stack and initialized with its "car" and "cdr" components. As we have already pointed out, however, returning a value typically causes its eviction (and the eviction of its components). Therefore, although `CONS` tries to be lazy, the effect of returning the newly allocated object causes its eviction to the heap, so our `CONS` isn't lazy after all! If we call this `CONS` with an explicit continuation, however, within which the newly allocated list cell is manipulated in a normal fashion, then the cell is not immediately evicted, and remains lazy.

Below is such an implementation of a lazy `CONS` in C.[20]

```
void lazy_cons(x,y,cont)
  int x; list y; void cont(list);
  {struct {int car; list cdr;} z;    /* The cons cell. */
   z.car=x; z.cdr=y; /* Initialize the lazy cons cell. */
   cont(&z);}          /* Give cont ptr. to cons cell. */
```

The use of continuation-passing-style allows the programmer himself to choose whether allocation will be lazy or not. In this way, he can use his greater knowledge of the program behavior to avoid unnecessary evictions, but also avoid the creation of large amounts of garbage in the stack.

Continuation-passing style has yet another benefit. Unlike normal nested function-application notation, continuation-passing style can deal with multiple returned values. For example, a Euclidean division algorithm "function" can return both a quotient and a remainder. In such a case, the "continuation" function must utilize more than one parameter in order to receive all of the results. Common Lisp also provides a number of forms to handle "multiple values", but does not utilize continuation-passing style for their implementation. Common Lisp multiple values are not strictly necessary, as all of the benefits of multiple values can be achieved through the composing of multiple values into a Lisp structure which can then be decomposed by the user of the function. In order to save the time and garbage collection required to compose and decompose these structures, however,

[16] Result expectation is the run-time analogue of a classic compiler optimization used in expression-oriented languages like Lisp.

[17] Although values are preserved on the stack, exited stack frames are spliced out of the call-chain so that `unwind-protect`'s in exited frames are not inadvertently executed.

[18] The MIT Lisp Machine function call instruction uses a similar coding (without lazy allocation) for its "destination operand" [Greenblatt74]; however, the callee does not utilize this information to avoid allocating useless results!

[19] The "continuation-passing style" (CPS) of programming, as discussed in the next section, can achieve the same allocation behavior as result expectation, but with greater overhead. Furthermore, one cannot use CPS on code for which one does not have the source—e.g., library code—so result expectation is to be preferred.

[20] Due to the lack of full function closures in C, we would have trouble actually using this `CONS` in any serious way.

Common Lisp uses a special mechanism to provide multiple values. We show that the benefits (including stack allocation) of Common Lisp's multiple-value mechanism can be simulated through a new first-class "multiple-value" structure type together with lazy allocation.[21]

```
(defstruct multiple-value
  (values nil :read-only t))

(defun values (&rest args)
  (make-multiple-value :values args))
(defun multiple-value-call (fn &rest args)
  (apply fn
         (mapcan
           #'(lambda (arg)
                (if (multiple-value-p arg)
                  (multiple-value-values arg)
                  (list arg)))
           args)))
```

To provide the programmer with the retention benefits of CPS without the requirement of turning his source code inside-out, we define the `CONTCALL` special form. The semantics of `CONTCALL` can be defined as follows:

```
(defun contcall (continuation fn &rest args)
  (multiple-value-call continuation (apply fn args)))
```

In other words, `CONTCALL` applies the function to the arguments, and than applies the "continuation" function to this result. The implementation of `CONTCALL` is special, however. Whereas the stack would normally have been contracted after the execution of `(apply fn args)`, it is not, so that the result(s) of this application is (are) left on the stack for the application of `continuation`. Using `CONTCALL`, we can then define Common Lisp's `MULTIPLE-VALUE-BIND` special form, whose purpose appears to be the extraction of multiple values from a function call *without causing any extraneous heap allocation*.

```
(defmacro multiple-value-bind
          (vars (fn . args) &body body)
  `(contcall #'(lambda ,vars ,@body) ,fn ,@args))
```

Of course, once we have lazy allocation and `CONTCALL`, we no longer need to clutter up the Lisp language with "multiple values", since the "non-consing" benefits can already be achieved without multiple values. `CONTCALL` can also be used for a definition of a `LET` which extends the stack so that the variables are bound to stack-allocated values. Such a stack-extending `LET` is usually the intention of the programmer; this is the motivation for proposals for a "`dynamic-let`" [Queinnec88], but without causing failure if the values escape the scope of the allocation.

```
(defmacro dlet ((var (fn . args)) &body body)
  `(contcall #'(lambda (,var &rest ignore) ,@body)
             ,fn ,@args))
```

Below, we show how to program a complex division routine which allocates all intermediate results on the stack.

```
(defun cdiv (z1 z2)
  (dlet ((z2bar (conjugate z2)))
   (dlet ((z2norm (ctimes z2 z2bar)))
    (dlet ((z1z2bar (ctimes z1 z2bar)))
     (dlet ((rz2norm (realpart z2norm)))
      (complex (/ (realpart z1z2bar) rz2norm)
               (/ (imagpart z1z2bar) rz2norm)))))))
```

Using continuation-passing style to extend the life of stack-allocated objects can be used for more substantial applications. For example, the storage needed for the intermediate results in a chain of matrix multiplications can be allocated in this fashion, so that only the final product matrix becomes a first-class heap object.

K. "Functional" Data Structures

So far, our lazy allocation model has utilized strict "relocation" semantics in order to preserve the "object identity" of allocated objects. In this semantics, there is only one "true" location for each object, but this location can sometimes change. For "functional" data structures—data structures which cannot be side-effected—we can relax the strict "relocation" semantics and utilize "copying" semantics. This is because the behavior of a functional data structure is determined by the values of its components, and since they cannot be changed, a copy of the data structure having the same components will have the same behavior. (A more thorough treatment of object identity for functional objects can be found in a companion paper [Baker93].)

[21] This multiple-value structure should be *functional* [Baker93] to achieve the maximum benefits of lazy allocation.

Copying semantics for functional objects can have some benefits in our lazy allocation model. When a functional object is copied, one need not necessarily leave a forwarding address, since the original is as good as the copy. Since forwarding addresses must be detected and followed during execution, the cost of detection and following may be more than the costs of copying. At the hardware level, the installation of forwarding pointers also causes a cache write-back, which adds additional load to the memory system. Thus, for small or unshared functional objects the cost of copying is less than the cost of storing, checking and following forwarding pointers.

Copying semantics can be exponentially less efficient than relocation semantics if substantial substructure sharing occurs, however. A simple linear Lisp list of length $\mathbf{n}$ in which the CAR of each list cell is assigned to be the same as its CDR has been called a "blam list" [McCarthy65] because it explodes into a structure of $2^{\mathbf{n}}$ list cells when it is functionally copied. The only exponential blowup of this type we have observed occurs in Macsyma's representation of the determinant of an $\mathbf{n}\times\mathbf{n}$ matrix in $O(\mathbf{n}^3)$ cells; this structure expands into an expression with $O(\mathbf{n}!)$ terms. On the other hand, the extended size of a functional data structure is constant and can be computed incrementally as it is constructed. The information needed to make a copy/no-copy decision can therefore be gathered cheaply at run-time.

There are a number of "functional" structures even in imperative languages. Argument lists and functional arguments are usually side-effect free. In some languages, character strings cannot be modified by side-effects. ANSI C offers the `const` qualifier. Scheme continuation structures, being similar to functional arguments, are side-effect free, although they may have pointers to non-functional objects. The various kinds of numbers in Common Lisp are functional—even large objects like infinite precision integers and structured objects like complex floating point numbers. The "multiple values" structures returned from Common Lisp function calls are also functional, even though they are not first-class Common Lisp objects.

The functionality of these objects—at least their top level structures—accounts for many of the other variations on lazy allocation. Thus, while MacLisp used lazy allocation for integers and floating point numbers [Steele77], it did not have to leave or check for forwarding addresses because these numeric objects were functional. Similarly, lazy allocation implementations of functional arguments and continuations do not bother to leave or check for forwarding addresses because there is very little potential sharing, and evictions happen very rarely, so copying is not a problem.

Due to Common Lisp's insistence upon the use of true Lisp lists for `&REST` arguments, however, one cannot legally use copying semantics for these objects, because Common Lisp list cells can be side-effected. As a result, the lazy allocation of `&REST` arguments is less efficient than if `&REST` arguments were based on a *functional* sequence structure instead of non-functional list cells. Scheme's requirement that `&REST` lists be *always* copied is just as bad, because the majority of such lists are functional and could otherwise be shared and thereby avoid a quadratic explosion of copying in deeply nested recursions [Dybvig87].

6. Future Work — An Incremental Model

The model as described above is not particularly incremental, because a transporter trap in a deeply nested stack frame could cause an unbounded number of objects to be copied before returning to the execution of the program. One can view the copying effort involved in eviction as the effort which was *deferred* by lazy allocation, and has suddenly come due. While this effort might still be less than the amount of effort saved by using lazy allocation, it is time that is not easily interrupted, and can therefore cause problems in a real-time system.

A more incremental system would evict objects from a stack frame just before it is popped, and would evict only the "top level" of those objects. Unfortunately, this sort of a system leads to great complications. In such a system, a stack frame must now be scanned before popping, in order to evict any remaining objects. However, unlike the non-incremental scheme where we could inductively prove that no pointers to the stack frame exist at the time of popping, the incremental scheme has no such property. If there are live objects remaining in the stack frame, then *ipso facto* there must be live pointers. Unfortunately, we do not know where those pointers are, so we cannot update them when the objects are moved out of the stack frame.

The only solution is to follow a technique invented by Bishop [Bishop77] and used by Lieberman and Hewitt [Lieberman83]. We use a separate *entry* table to keep track of those pointers which violate a stack's well-ordering. This entry table initially starts out empty, but when a stack frame is cleared of live objects, some of these objects may continue to point into other stack frames. These pointers are routed indirectly through the entry table. When the next stack frame is to be cleared, the entry table is searched for objects entering the stack frame, and those objects are then relocated. In this way, we can incrementalize the eviction process.

Our incremental scheme is nearly equivalent to Lieberman and Hewitt's *generational garbage collection*, in which the global heap and each stack frame are separate generations. Unlike Lieberman and Hewitt, however, who would move a result object through every intermediate generation, we move such objects directly to the oldest generation—the global heap—in order to avoid a quadratic explosion in copying effort [Dybvig88]. Our policy is similar to Ungar's *tenuring* policy [Ungar84], which also avoids the copying of long-lived objects through the intermediate generations. The address ordering relation, the relocation process and the manipulation of the entry tables in our scheme are all identical to that of Lieberman and Hewitt, however.

7. Conclusions and Previous Work

We have shown how a general model called *lazy allocation* can simply and elegantly explain many traditional programming language optimizations aimed at increasing the fraction of storage allocations that can be performed on a stack. Lazy allocation requires the ability to deal with objects which can be suddenly moved, but once this cost has been paid, lazy allocation can result in great simplifications in other parts of a language implementation. Lazy allocation puts most of its overhead burden on assignments, which makes it attractive for the "mostly functional" programming styles of modern expression-oriented languages. Lazy allocation also has benefits in shared-memory multi-processor environments where the potential bottleneck of a global allocator is shielded by lazy stack allocation from the bulk of the allocation load.

We have also described a new run-time technique called *result expectation*, which informs called functions of what results are expected and where they should be put, so that unexpected results need not be heap-allocated. While interesting in its own right, result expectation works with lazy allocation to reduce the number of evictions of function results from functions called for their effect rather than for their result.

The concept of lazy allocation is the result of 10 years of pondering the possibility of "backing up"—under certain circumstances—the allocation pointer of the author's real-time garbage collection algorithm [Baker78a], in order to improve its amortized performance. The single-bit reference count [Wise77] for stack-allocated objects can be subsumed by address ordering, yielding the current concept of lazy allocation.

Due to the ubiquity of the problem, the literature on stack-allocating various kinds of objects is so large that we can reference only a small fraction. The stack allocation of variable binding environments encompasses the Algol-60 *display* [Randell64], Lisp's binding *environments* [Greenblatt74], Lisp's *shallow binding* [Baker78b], Lisp's *cactus stacks* [Bobrow73]. The stack allocation of functional objects like numbers is discussed in [Steele77] and [Brooks82]. "Dynamic extent objects" [Queinnec88] have been proposed as part of the Eu_Lisp standard. Our lazy allocation completely subsumes dynamic extent objects, and our trapping for the purpose of eviction is no more expensive than trapping to determine lifetime errors.

The deletion (stack-allocation) versus retention (heap-allocation) implementation strategies for Algol-like compiled languages has been studied by [Berry71] [Fischer72] [Berry78a] [Berry78b] [Berry78c]. [Blair85] describes an *optimistic stack-heap*, which is approximately our lazy allocation applied to stack frames; unlike our lazy allocation, however, the optimistic stack-heap is not used for user-allocated data. In other words, these models do not separate the issues of language implementation (frames) from storage allocation (stack allocation).

The stack allocation of functional arguments has been studied by [Johnston71], [Steele78], [McDermott80] and many others. Johnston [Johnston71] is said to have used the term *lazy contour* which is a close approximation to our lazily allocated stack frame.

The stack allocation of continuations has been studied by Steele [Steele78], Stallman [Stallman80], Bartley [Bartley86], Clinger [Clinger88], Danvy [Danvy87], Deutsch and Schiffman [Deutsch84], Dybvig [Dybvig87], Kranz [Kranz86] [Kranz88], [Moss87], [Hieb90] and many others. The straight-forward application of lazy stack allocation to Dybvig's heap model [Dybvig87] yields a large fraction of the optimizations he performs by hand; the lazy allocation of continuations also avoids the multiple stack copies mentioned in [Hieb90]. Deutsch and Schiffman use the term *volatile* for lazy stack frames, and *stable* for evicted stack frames.

Dybvig [Dybvig88] is apparently the first to have pointed out in print that the consistent copying of a large argument to successive levels of recursion can convert a linear algorithm into a quadratic one.

The concept of lazy allocation was almost discoved by Lieberman and Hewitt [Lieberman83], since they had all of the necessary machinery. However, the additional concept of "genetic order" [Terashima78] was missing. McDermott discovered a form of lazy allocation [McDermott80] for implementing the variable-binding environments used in a lexically-scoped Lisp interpreter, and he also indicated its possible use for managing Scheme continuations. [Morrison82] is simply lazy allocation applied to the consing performed in [Baker78b]! [Mellender89] implements Smalltalk with a scheme based on the same concepts as "lazy allocation", but with substantially greater complexity. Tucker Taft [Kownacki87] [Taft91] independently developed for the Ada-9X language the idea of a run-time "scope check", with a user-defined copy-to-heap if required; this excellent proposal was unfortunately later withdrawn.

Stallman's *phantom stacks* [Stallman80], which were invented to implement Scheme on the MIT Scheme Chip [Steele79], are an interesting alternative to solving the same kinds of stack allocation problems as lazy allocation. In phantom stacks, objects are stack-allocated in the same manner as in lazy allocation. The difference between the two models comes when LIFO order is violated. In lazy allocation, we evict objects from the stack to the heap, while in phantom stacks, a new stack is initiated, the old stack is abandoned, in place, where it becomes a passive set of objects in the heap. Thus, lazy allocation and phantom stacks are duals of one another: lazy allocation moves objects from the stack, while phantom stacks moves the stack from the objects. [Hieb90] rediscovered phantom stacks and gives an analysis which is more appropriate for the execution of Scheme on a modern RISC processor.

Both *Prolog* [Warren83] and *Forth* [Moore80] make more extensive use of stacks than do traditional Lisp implementations, and gain substantially in elegance and speed as a result.

8. Acknowledgements

We wish to thank Dan Friedman, Andre van Meulebrouck, Carolyn Talcott and the referees for their helpful suggestions and criticisms.

9. References

Aho, A.V. "Nested stack automata". *JACM* 16,3 (July 1969),383-406.

ANSI-C. *Draft Proposed American National Standard Programming Language C.* ANSI, New York, NY, 1988.

Appel, Andrew W. "Garbage Collection Can Be Faster Than Stack Allocation". *Info. Proc. Let. 25* (1987),275-279.

Appel, A.; MacQueen, D.B. "A Standard ML Compiler". *ACM Conf. Funct. Prog. & Comp. Arch.*, Sept. 1987.

Appel, Andrew W.; Ellis, John R.; and Li, Kai. "Real-time concurrent garbage collection on stock multiprocessors". *ACM Prog. Lang. Des. and Impl.*, June 1988,11-20.

Appel, Andrew W. "Simple Generational Garbage Collection and Fast Allocation". *SW Prac. & Exper. 19*,2 (Feb. 1989),171-183.

Baker, Henry G. "List Processing in Real Time on a Serial Computer". *CACM 21,4* (April 1978), 280-294.

Baker, Henry G. "Shallow Binding in Lisp 1.5". *CACM 21*,7 (July 1978),565-569.

Baker, Henry G. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Progr.*, June 1990,218-226.

Baker, Henry G. "Equal Rights for Functional Objects". ACM OOPS Messenger 4,4 (Oct. 1993), 2-27.

Barth, J. "Shifting garbage collection overhead to compile time". *CACM* 20,7 (July 1977),513-518.

Bartley, D.H., Jensen, J.C. "The Implementation of PC Scheme". *ACM Lisp & Funct. Prog.*, Aug. 1986, 86-93.

Berry, D.M. "Block Structure: Retention vs. Deletion". *Proc. 3rd Sigact Symp. Th. of Comp.*, Shaker Hgts., OH, 1971.

Berry, D.M., *et al.* "Time Required for Reference Count Management in Retention Block-Structured Languages, Part 1". *Int'l. J. Computer & Info. Sci. 7*,1 (1978),11-64.

Berry, D.M., *et al.* "Time Required for Reference Count Management in Retention Block-Structured Languages, Part 2". *Int'l. J. Computer & Info. Sci. 7*,2 (1978),91-119.

Berry, D.M., and Sorkin, A. "Time Required for Garbage Collection in Retention Block-Structured Languages". *Int'l. J. Computer & Info. Sci. 7*,4 (1978),361-404.

Bishop, P.B. *Computer Systems with a very large address space and garbage collection*. Ph.D. Thesis, TR-178, MIT Lab. for Comp. Sci., Camb., MA, May 1977.

Blair, J.R., Kearns, P., and Soffa, M.L. "An Optimistic Implementation of the Stack-Heap". *J. Sys. & Soft. 5* (1985),193-202.

Bobrow, D.G., and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments". *CACM 16*,10 (Oct. 1973),591-603.

Bobrow, et al. "Common Lisp Object System Specification X3J13", *ACM SIGPLAN Notices*, v.23, Sept. 1988; also X3J13 Document 88-002R, June 1988.

Boehm, H.-J., Demers, A. "Implementing Russell". *Proc. Sigplan '86 Symp. on Compiler Constr., Sigplan Not. 21*,7 (July 1986),186-195.

Brooks, R.A., *et al*. "An Optimizing Compiler for Lexically Scoped LISP". *ACM Lisp & Funct. Prog. 1982*,261-275.

Brooks, R.A. "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware". *ACM Lisp & Funct. Progr. 1984*,256-262.

Chase, David. "Garbage Collection and Other Optimizations". PhD Thesis, Rice U. Comp. Sci. Dept., Nov. 1987.

Cioni, G., Kreczmar, A. "Programmed Deallocation without Dangling Reference". *IPL 18*, 4 (May 1984),179-187.

Clinger, W.D., Hartheimer, A.H., and Ost, E.M. "Implementation Strategies for Continuations". *ACM Conf. on Lisp and Funct. Prog.*, July 1988,124-131.

CLtL: Steele, Guy L., Jr. *Common Lisp: The Language*. Digital Press, 1984.

Danvy, O. "Memory Allocation and Higher-Order Functions". *Proc. Sigplan '87 Symp. on Interpreters and Interpretive Techniques, ACM Sigplan Notices 22*,7 (July 1987),241-252.

Dybvig, R.K. "Three Implementation Models for Scheme". Ph.D. Thesis, Univ. N. Carolina at Chapel Hill Dept. of Comp. Sci., also TR#87-011, April 1987,180p.

Dybvig, R.K. "A Variable-Arity Procedural Interface". *ACM Lisp and Functional Prog*. (July 1988),106-115.

Fischer, M.J. "Lambda Calculus Schemata". *Proc. ACM Conf. on Proving Asserts. re Progs.*, *Siglan Not. 7*,1 (Jan. 1972).

Fisher, D.A. "Bounded Workspace Garbage Collection in an Address-Order Preserving List Processing Environment". *Inf.Proc.Lett. 3*, 1 (July 1974),29-32.

Friedman, D.P., and Wise, D.S. "CONS Should not Evaluate its Arguments". In S. Michaelson and R. Milner (*eds*.), *Automata, Languages and Programming*, Edinburgh U. Press., Edinburgh, Scotland, 1976,257-284.

Goldberg, B., and Park, Y.G. "Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations". *Proc. ESOP'90*, Springer-Verlag, May 1990.

Greenblatt, R., *et al*. "The Lisp Machine". AI WP 79, MIT, Camb., MA, Nov. 1974, rev. vers. in Winston, P.H., and Brown, R.H., *eds. Artificial Intelligence, an MIT Perspective: Vol. II*. MIT Press, Camb., MA, 1979.

Harbison, S.P., and Steele, G.L., Jr. *C: A Reference Manual, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

Harrington, Steven J. "Space efficient copying storage recovery". *Computer J. 24*,4 (1981),316-319.

Hederman, Lucy. "Compile Time Garbage Collection". MS Thesis, Rice U. Computer Science Dept., Sept. 1988.

Hieb, R., Dybvig, R.K., and Bruggeman, Carl. "Representing Control in the Presence of First-Class Continuations". *ACM Sigplan '90 Conf. Prog. Lang. Design & Impl.*, June 1990,66-77.

IEEE-Scheme. *IEEE Standard for the Scheme Programming Language*. IEEE-1178-1990, IEEE, NY, Dec. 1990.

Johnston, J.B. "The Contour Model of Block Structured Processes". *Sigplan Notices* (Feb. 1971).

Jouppi, N.P. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers". *Proc. 17th Int'l. Symp. on Computer Arch., ACM Computer Arch. News 18*,2 (June 1990).

Kownacki, R., and Taft, S.T. "Portable and Efficient Dynamic Storage Management in Ada". *Proc. ACM SigAda Int'l. Conf. "Using Ada"*, Dec. 1987,190-198.

Kranz, D., et al. "ORBIT: An Optimizing Compiler for Scheme". *Proc. Sigplan '86 Symp. on Compiler Constr., Sigplan Notices 21*,7 (July 1986),219-233.

Kranz, David A. ORBIT: An Optimizing Compiler for Scheme". Ph.D. Thesis, Yale Univ., also YALEU/DCS/RR-632, Feb. 1988,138p.

Lieberman, H., Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*, 6 (June 1983),419-429.

McCarthy, J., *et al*. *LISP 1.5 Programmer's Manual*. MIT Press, Camb., MA, 1965.

McDermott, D. "An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-scoped LISP". *1980 Lisp Conf.*, Stanford, CA, Aug. 1980,154-162.

Mellender, F, *et al*. "Optimizing Smalltalk Message Performance". In Kim, W., and Lochovsky, F.H., *eds.*, *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, Reading, MA, 1989,423-450..

Moon, D. "Garbage Collection in a Large Lisp System". *ACM Symp. on Lisp & Funct. Prog.*, 1984, 235-246.

Moore, Charles H. "The Evolution of FORTH, an Unusual Language". *Byte Magazine 5*,8 (special issue on the Forth Programming Language) (Aug. 1980),76-92.

Morrison, Donald F, and Griss, Martin, L. "An Efficient A-list-like Binding Scheme". Unpublished manuscript, Dept. of Computer Science, U. of Utah, Jan. 1982,12p.

Moses, J. "The function of FUNCTION in Lisp". Memo 199, MIT AI Lab., Camb., MA, June 1970.

Moss, J.E.B. "Managing Stack Frames in Smalltalk". *SIGPLAN '87 Symp. on Interpreters and Interpretive Techniques*, in *Sigplan Notices 22*,7 (July 1987), 229-240.

Queinnec, Christian. "Dynamic Extent Objects". *Lisp Pointers 2*,1 (July-Sept. 1988),11-21.

Randell, B., and Russell, L.J. *ALGOL 60 Implementation*. Acad. Press, London and NY, 1964.

Rees, J. and Clinger, W., *et al*. "Revised Report on the Algorithmic Language Scheme". *Sigplan Not. 21*,12 (Dec. 1986),37-79.

Richards, M., Evans, A., and Mabee, R.R. "The BCPL Reference Manual". MIT MAC TR-141, Dec. 1974,53p.

Ruggieri, C.; Murtagh, T.P. "Lifetime analysis of dynamically allocated objects". *ACM POPL '88*,285-293.

Samples, A.D., Ungar, D, and Hilfinger, P. "SOAR: Smalltalk without Bytecodes". *OOPSLA '86 Proc., Sigplan Not. 21*, 11 (Nov. 1986),107-118.

Sandewall, E. "A Proposed Solution to the FUNARG Problem". 6.894 Class Notes, MIT AI Lab., Sept. 1974,42p.

Shaw, Robert A. "Improving Garbage Collector Performance in Virtual Memory". Stanford CSL-TR-87-323, March 1987.

Sherman, Mark, *et al*. "An ADA Code Generator for VAX 11/780 with UNIX". *ACM SIGPLAN ADA Conf.*, *SIGPLAN Notices 15*,11 (Nov. 1980),91-100.

Stallman, Richard M. "Phantom Stacks: If you look too hard, they aren't there". AI Memo 556, MIT AI Lab., 1980.

Stanley, T.J., and Wedig, R.G. "A Performance Analysis of Automatically Managed Top of Stack Buffers". *Proc. 14th Int'l. Symp. on Computer Arch., ACM Computer Arch. News 15*,2 (June 1987),272-281.

Steele, Guy L., Jr. "Fast Arithmetic in Maclisp". *Proc. 1977 Macsyma User's Conference*, NASA Sci. and Tech. Info. Off. (Wash., DC, July 1977),215-224. Also AI Memo 421, MIT AI Lab., Camb. Mass.

Steele, Guy L., Jr. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*. AI-TR-474, Artificial Intelligence Laboratory, MIT, May 1978.

Steele, Guy L., Jr., and Sussman, Gerald J. "Design of LISP-Based Processors or SCHEME: A Dielectric LISP or, Finite Memories Considered Harmful or, LAMBDA: The Ultimate Opcode". MIT AI Memo 514, March 1979.

STEELMAN. Dept. of Defense Requirements for High Order Computer Programming Languages. June 1978.

Taft, S. Tucker, *et al. Ada-9X Draft Mapping Document*. Wright Lab., AFSC, Eglin AFB, FL, Feb. 1991.

Terashima, M., and Goto, E. "Genetic Order and Compactifying Garbage Collectors". *IPL 7*,1 (Jan. 1978),27-32.

Unger, D. "Generation Scavenging: A non-disruptive, high performance storage reclamation algorithm". *ACM Soft. Eng. Symp. on Prac. Software Dev. Envs., Sigplan Notices 19*,6 (June 1984),157-167.

Warren, David H.D. *Applied Logic—Its Use and Implementation as a Programming Tool*. Ph.D. Thesis, U. of Edinburgh, 1977, also Tech. Note 290, SRI Int'l., Menlo Park, CA, 1983,230p.

Wise, D.S., and Friedman, D.P. "The One-Bit Reference Count". *BIT 17* (1977),351-359.

