ACM *Sigplan Notices* **29**, 9 (September 1994), 38-43.

Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures

Henry G. Baker

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

This material is based upon work supported by the National Science Foundation under Grant No. III-9261682.

Reference counting can be an attractive form of dynamic storage management. It recovers storage promptly and (with a garbage stack instead of a free list) it can be made "real-time"—i.e., all accesses can be performed in constant time. Its major drawbacks are its inability to reclaim cycles, its count storage, and its count update overhead. Update overhead is especially irritating for functional (read-only) data where updates may dirty pristine cache lines and pages.

We show how reference count updating can be largely eliminated for functional data structures by using the "linear style" of programming that is inspired by Girard's linear logic, and by distinguishing normal pointers from *anchored pointers*, which indicate not only the object itself, but also the depth of the stack frame that anchors the object. An *anchor* for a pointer is essentially an enclosing data structure that is temporarily locked from being collected for the duration of the anchored pointer's existence by a deferred reference count. An *anchored pointer* thus implies a reference count increment that has been deferred until it is either cancelled or performed.

Anchored pointers are generalizations of "borrowed" pointers and "phantom" pointers. Anchored pointers can provide a solution to the "derived pointer problem" in garbage collection.

INTRODUCTION

Reference counting [Collins60] can be an attractive form of dynamic storage management for functional (read-only) data structures because the cycles that cause trouble for reference counting do not exist in these data structures. Unfortunately, reference counting extracts a penalty not only when creating or destroying data structures, but also when simply traversing them. For example, when the length of a list is computed, the counts of the cells along the list are each incremented and then decremented. Although this traversal results in no net change to any count, the work to perform 2n count updates for a list of length n is substantial. In a modern RISC architecture with a data cache, these count references may cause lines to be brought into the cache that would not otherwise be touched, and worse still, these lines are also "dirtied", requiring that they be rewritten—even though the data eventually written out is identical to that brought in! In a shared-memory multiprocessor system, updates to counts require synchronization in order to avoid losing or gaining counts due to conflicting reads and writes. Thus, while reference count updating for the creation and deletion of structures may be tolerable, this continual updating for simple traversals is not.

In a classical reference count system, the reference count on a node indicates the exact number of references—whether from other nodes or local variables—that currently point to the node. The copying of a reference causes a reference count increment and the deletion of a reference causes a reference count decrement. The "linear style" of programming inspired by linear logic (see Appendix I) helps to minimize the reference count updating that is caused by the copying and deletion of references from local variables. In the linear style, the policy is that variable references are destructive reads—i.e., `cons(a,b)` does not have to change the reference counts on `a`,`b` because these references have simply moved from local variables into storage. The linear style makes the creation and deletion of references explicit—duplicating a local reference requires calling the function `dup`, while deleting a local reference requires calling the function `kill`. The function `dup` increments the reference count of the target object, while the function `kill` decrements the reference count, and may also reclaim the object, recursively killing its component references.

While the linear style can help organize and reduce reference count updating, it does not eliminate the updating that results from simple traversal. Thus, a linear `length` function will still increment and decrement reference counts on every cell it traverses. To understand this better, we show the code for a linear `length`:

```
(defun length (x)                                      ; Length function in linear style.
  (if-null x (progn (kill x) 0)                                  ; Kill x reference to nil.
    (dlet* (((carx . cdrx) x))                              ; Dissolve x into carx, cdrx.
      (kill carx)1                                               ; Kill reference to carx.
      (1+ (length cdrx)))))                  ; Our result is one more than length of cdrx.
```

When `length` is entered, the reference count on `x` is almost certainly >1, or else `length` will consume its argument and return it to the freelist! Let us assume that the reference count of the head of the list is 2 and the rest of the list is

[1]Following logic languages, we could use syntax like `dlet* (((_ . cdrx) x))` to immediately kill the `car` of `x`.

unshared. A standard `dlet*` adds references to `carx`, `cdrx` and then deletes the reference to `x`. The reference to `carx` is immediately killed, and the reference to `cdrx`, which now has a reference count of 2, is passed to a recursive invocation of `length`. At list end, the extra reference from `x` to `nil` is killed, and `0` is returned. Thus, `length` performs 4n count updates (each *element* of the list has its reference count updated twice) for an n-element list.

Since `length` eventually returns the reference counts to their initial state, what would happen if we simply avoided all updates? A problem arises because the second and succeeding nodes on the list have reference counts of 1—the `dlet*`'s will reclaim all of these nodes and put them onto the freelist! We must therefore somehow suspend reclamation while `length` is in progress. Suppose that we use *two* types of pointers—a *normal* pointer and a *deferred increment* pointer. The normal pointer acts as we have described above, while the deferred increment pointer signals that reclamation has been turned off. In other words, *a deferred increment pointer is a pointer with a deferred reference count increment*. If we copy a deferred increment pointer using `dup`, we simply copy the bits, because n copies of a deferred increment implies a deferred increment of n. Similarly, if we delete a deferred increment pointer using `kill`, we simply delete the reference, since the decrement associated with `kill` cancels the deferred increment. When `cons(x,y)` is performed on a *normal* pointer `x`, the reference count on `x` does not have to be adjusted since the reference is simply transferred from the local variable `x` to the cons cell. However, when `cons(x,y)` is performed on a *deferred increment* pointer `x`, the reference count on `x` *does* have to be incremented, since a deferred reference means a deferred increment, and components of data structures have to be normal. The treatment of `dlet*` for a deferred increment pointer `x` is also elegant: the deferred increment on `x` cancels with `dlet*`'s decrement of `x`, and the local variables `carx`, `cdrx` are also deferred, so their increments are deferred. So deferred increment pointers appear to elegantly achieve our goals—executing `length` on a deferred increment pointer `x` performs no reference count updates!

If deferred increment pointers are so efficient, why not eliminate normal pointers altogether? The problem with this proposal is that no storage can *ever* be recovered with only deferred increment pointers—i.e., we are back in the realm of tracing garbage collection. Consider `kill(cons(x,y))`. If `cons` returns a deferred increment pointer, then `kill` does nothing, and the cons cell is lost. Thus, `cons` itself must always return a normal (non-deferred) pointer.[2]

We therefore propose a system of pointers that are dynamically typed as normal/deferred-increment.[3] Unfortunately, this system still does not quite work, as the `third` function demonstrates:

```
(defun third (x)                                   ; Linearly return the third cons in list x.
  (dlet* (((carx . cdrx) x)                                          ; Dissolve the first cons.
          ((cadrx . cddrx) cdrx))                                    ; Dissolve the second cons.
    (kill carx) (kill cadrx) cddrx))           ; Kill first 2 elements, then return 3rd cons.
```

If `third` is passed a deferred increment pointer `x`, it will traverse the list `x` without changing reference counts until it gets to the third cons cell. It then returns a deferred increment pointer to this third cons cell. Unfortunately, the caller to `third` can now kill the list, in which case the third cons will also be reclaimed, because the increment implied by the deferred pointer to it will never have been performed. We can fix this bug by making the policy that *only normal pointers can be returned from a function*. With this policy, the returned value from `third` will be coerced back to a normal pointer by performing the increment implied by the deferred pointer.[4] This policy will work correctly, because it is somewhat more prompt than the correct "anchored pointer" scheme described below, but is also more expensive, because it does not defer reference count updates as long as possible. However, if we have but one pointer bit to give to the cause, this normal/deferred scheme is still better than a classical reference count scheme—e.g., functions like `length` "win completely" with just a one-bit distinction.

A reference count updating scheme lazier than deferred increment pointers can be obtained at the cost of additional pointer bits by means of *anchored pointers*. An anchored pointer is a deferred increment pointer with an additional component which indicates its *anchor*. One implementation might use an integer component indicating the level in the stack at which the object is *anchored*. When a functional cell pointed at by a deferred increment pointer `x` is dissolved into its components, they each *inherit* the level number associated with `x`. An anchored pointer not only indicates deferredness, but also indicates the extent during which this deferral is valid. If a deferred increment pointer is returned from stack level n, it must have a level number that is strictly less than n. In other words, when returning

[2]The Deutsch-Bobrow reference count scheme [Deutsch76] [Deutsch80] does not count references from local variables, and thereby avoids these count updates. Their scheme requires a stack scan before reclaiming storage, however, which scan is nearly impossible to do on stacks formatted by optimizing compilers for modern RISC architectures.

[3]The *normal/deferred-increment* distinction is essentially identical to the *real/phantom* distinction of [Lemaître86] and the *owned/borrowed* distinction of [Gelernter60].

[4][Lemaître86] calls this normalization operation "materialization".

a deferred increment pointer from stack level n+1, one first checks to see that the level is ≤n, and if not, the pointer is coerced to normal—i.e., the reference count increment is performed.

The implementation of the anchored pointer scheme can be confined to a `dlet1` special form, which dissolves a single list cell and binds its components. We identify the anchor level with the index of this `dlet1` in the stack—i.e., each `dlet1` form has its own level number. If `dlet1` is given a normal pointer, then it checks the reference count. If the count is 1, then the cell is unshared, and `dlet1` has the only pointer to this cell. The car and cdr are bound as normal pointers, and the cell is recycled. If the count is >1, then the cell is shared, so the car and cdr are bound as anchored pointers with a level number being the current `dlet1` level. Later, when `dlet1` must return values, these values are checked for deferral and normalized if their level numbers are ≥ the current level. Finally, the reference count on the cell input to the `dlet1` is decremented. The last case involves an anchored pointer input to `dlet1`. In this case, the car and cdr bindings are also anchored with the same level number as their parent pointer. Since the parent pointer must have been anchored at a level strictly less than the current level, the return values need not be checked at all, since they will either already be normal, or will not require normalization until a lower level.

Working with anchored pointers can be extremely efficient. In the Boyer Benchmark [Gabriel85], for example, the (functional) database of rewrite rules is bound at "top-level"—i.e., stack level 0. Thus, traversals of these data structures can be performed entirely by anchored pointers without updating reference counts—an extremely important characteristic for a shared-memory multiprocessor implementation of this benchmark, where multiple processors have to concurrently access these shared read-only rules.

If our anchored pointer scheme is used in conjunction with a traditional tracing garbage collector, the deferred reference counts may give the collector fits. In order for the reference accounting to balance, a cell bound by a deferred `dlet1` should be given a *negative* deferred reference by the `dlet1`, and this negative deferred reference is finally normalized before the `dlet1` returns its values by causing its reference count to be decremented. Only `dlet1` and the garbage collector ever see these negative deferred references, however.

The inclusion of a level number with every pointer in a local variable is probably prohibitive on today's 32-bit architectures, because the additional information would double the number of registers required to store local variables. However, with the newer 64-bit architectures—e.g., MIPS, DEC—a 16-bit level number would not constrain addressability, and would avoid the requirement for additional registers. Checking and masking this level data, however, could present efficiency problems if the architecture is not organized for it.

'`WITH-ANCHORED-POINTER`' PROGRAMMING LANGUAGE CONSTRUCT

The `dlet1` special form described above works, but it may be more complex than strictly needed to solve the problem. What we really need is a form that is given a normal pointer and provides an anchored pointer for use only within a dynamic context. In other words, it puts the normal pointer into "escrow", and provides an anchored pointer for temporary use, and when the construct is exited and its values normalized, the escrowed pointer can then be dropped. We suggest something like the `with-anchored-pointer` special form:

```
(with-anchored-pointer (ap-name) (<expression>)   ; binds ap-name to copy of expression value.
  << use ap-name within this dynamic extent body >>
  < returned-value-expression >)
```

The `with-anchored-pointer` form evaluates `<expression>` to either a normal pointer or another anchored pointer. If it evaluates to an anchored pointer, then `with-anchored-pointer` acts just like a `let`-expression. However, if it evaluates to a normal pointer, then the pointer is saved internally, and an anchored pointer copy of the normal pointer is made and bound to the name `ap-name`. The body of the `with-anchored-pointer` form is evaluated with this new binding, and the values returned by the body are normalized if they depend upon the escrowed pointer. The normal pointer is killed just before the dynamic scope is exited, which involves decrementing its reference count.

Constructs like `with-anchored-pointer` allow the introduction of anchored pointers into a static type system. Within the body of the construct, the name is bound to a value which is guaranteed to be anchored, and thus many run-time checks may be omitted.

CONCLUSIONS AND PREVIOUS WORK

Programmers have been using temporary pointers to objects without updating their associated reference counts for decades—e.g., the Unix file system is reference counted, but it distinguishes between "hard" and "soft" links; soft links (created by `ln -s`) are uncounted. [Gelernter60] uses the distinction of *owned/borrowed* pointers; [Lemaître86] uses the concept of *phantom pointers*. Our concept of anchored pointers, however, generalizes these techniques and makes more precise the situations under which they are safe. Maclisp "PDL numbers" [Steele77] are similar to anchored pointers in that they encode a stack level and can be used safely only within a dynamic context. Since PDL numbers have no explicit reference count, however, the only way to normalize them is by copying them, which is done whenever they are stored into a more global environment or returned as a value of a function.

Other schemes involving 1-bit reference counts include [Wise77] [Stoye84] [Chikayama87] [Inamura89] [Nishida90] and [Wise92]. Anchored pointers can be particularly useful when a reference count cannot be incremented for some particular reason—e.g., it has only 1 bit [Wise77] [Chikayama87] [Inamura89] [Nishida90]. Our linear scheme for avoiding reference count updates has goals similar to those of [Deutsch76] [Deutsch80] and especially [Barth77] and [Park91]. In particular, our scheme defers reference count updating like Barth's and Park's in the hope of a decrement cancelling an increment, leaving no net change. However, ours is simpler than theirs—requiring only static linearity checking instead of global flow analysis. Furthermore, linear style makes avoiding count updates more likely.

Anchored pointers are similar to *generational reference counts* [Goldberg89] in that they both attempt to minimize count updates, both explicitly identify their generation and both use escape analysis. However, anchored pointers do not require a count field, and can be slightly lazier than generational reference counts. Anchored pointers are closely related to the schemes described in [Baker92CONS] and [Baker93Safe]. The scheme of [Baker93Safe], which has different goals and works for mutable data, stores its stack level numbers in the objects themselves rather than in the pointers to the objects. Some reference count schemes [Deutsch76] [Deutsch80] avoid reference count updating overhead during traversal by never counting references from local variables on the stack. Such schemes require that the stack be scanned before cells can be reclaimed. It is becoming apparent, however, that scanning the stack of a compiled language like C [Yuasa90] or Ada [Baker93Safe] is becoming increasingly difficult, making stack-scanning schemes infeasible.

Anchored pointers can also be used to improve the efficiency of tracing non-copying garbage collection. In a real-time non-relocating garbage collector such as [Baker92Tread], *anchored pointers can be accessed without a read barrier*. Furthermore, since an anchored pointer contains within it a proof that the target object is not garbage, *anchored pointers need not be traced by the garbage collector*. While tracing the pointer will reveal that the target object has already been marked, it is more efficient to not trace the pointer, so that the target never has to be paged into main memory or the cache. Anchored pointers are similar to `object` declarations in *Kyoto Common Lisp* [Yuasa90], because `object` declarations are not traced by the garbage collector. KCL `object` declarations are not safe, however, because they can refer to mutable data, and carry no proof of accessibility.

Anchored pointers are similar to *weak pointers*, in that they are not traced by a garbage collector, but are used for a completely different purpose. A weak pointer is expected to escape from the context in which its target is protected from reclamation, but when its target is reclaimed, the escaped pointer is cleared to `nil`, so the application will know that the object is gone. Anchored pointers, on the other hand, are used primarily for improving efficiency, and should have no user-visible semantic differences except for improved performance.

Anchored pointers can also provide a solution to the garbage collection problems of *derived pointers* [Ellis88] [Nilsen88] [Boehm91] [Diwan92]—i.e., pointers into the interior of a large object such as an array. [Ellis88] suggests that a derived pointer have the form <base-ptr,derived-ptr>; a normal pointer to an array element would have the form <base-ptr,index>. A derived pointer can be normalized into a normal pointer by computing index = (derived-ptr - base-ptr)/element-size. Our anchored pointers and their use in a dynamic extent can be considered a generalization of the static scheme proposed in [Boehm91]. Our anchoring scheme avoids the tracing of derived pointers in a non-copying collector.

Our reference count scheme is safe because it defers increments only within an extent where a cell cannot be reclaimed anyway. The non-prompt incrementation of reference counts means that a garbage collection that restores reference counts may restore the reference counts improperly. Our fix for this problem is to create *negative references* when a cons is dissolved by `dlet1`. This negative reference is normalized when `dlet1` exits. Negative references are also cleaned up when unwinding the stack—e.g., for C's `setjmp`/`longjmp`, or for Common Lisp's `catch`/`throw`.

Our reference count scheme is intended to support the efficient implementation of functional (read-only) objects—perhaps implemented by means of *hash consing* [Baker92BB] [Baker92LLL]—and so there is no attempt to handle mutable objects or cycles. The linear style of programming essentially requires functions to be *total*—i.e., they must return with a value. The explicit killing of references, and the normalization traps of anchored pointers requires that functions return in the normal way—the `catch`/`throw` of Common Lisp, the reified continuations of Scheme and the `setjmp`/`longjmp` of C will all cause significant problems with this storage management scheme.

ACKNOWLEDGEMENTS

Many thanks to David Wise and Kelvin Nilsen for their constructive criticisms on this paper.

REFERENCES

Abramsky, S. "Computational interpretations of linear logic". *Theor. Comp. Sci. 111* (1993), 3-57.
Baker, H.G. "CONS Should not CONS its Arguments". ACM *Sigplan Not. 27*,3 (March 1992), 24-34.
Baker, H.G. "The Treadmill: Real-Time Garbage Collection without Motion Sickness". *Sigplan Not. 27*,3 (1992).

Baker, H.G. "The Boyer Benchmark at Warp Speed". ACM *Lisp Pointers V*,3 (July-Sept. 1992), 13-14.
Baker, H.G. "Lively Linear Lisp — 'Look Ma, No Garbage!'". ACM *Sigplan Notices 27*,8 (Aug. 1992), 89-98.
Baker, H.G. "Safe and Leakproof Resource Management using Ada83 Limited Types". ACM *Ada Letters XIII*, 5 (Sep/Oct 1993), 32-42.
Baker, H.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.
Bekkers, Y., and Cohen, J., *eds*. *Memory Management: Proc. IWMM92* Springer LNCS 637, 1992.
Berry, G., and Boudol, G. "The chemical abstract machine". *Theor. Comp. Sci. 96* (1992), 217-248.
Boehm, H.-J. "Simple GC-Safe Compilation". GC Workshop at OOPSLA'91, Phoenix, AZ, Oct. 6, 1991.
Chikayama, T., and Kimuar, Y. "Multiple Reference Management in Flat GHC". *Logic Programming, Proc. 4th Intl. Conf.*, MIT Press, 1987, 276-293.
Chirimar, J., *et al*. "Proving Memory Management Invariants for a Language Based on Linear Logic". *Proc. ACM Conf. Lisp & Funct. Prog*., San Francisco, CA, June, 1992, also ACM *Lisp Pointers V*,1 (Jan.-Mar. 1992), 139.
Collins, G.E. "A method for overlapping and erasure of lists". *CACM 3*,12 (Dec. 1960), 655-657.
Deutsch, L.P., and Bobrow, D.G. "An efficient incremental automatic garbage collector". *CACM 19*,9 (Sept. 1976).
Deutsch, L.P. "ByteLisp and its Alto Implementation". *Lisp Conf*., Stanford, CA, Aug. 1980, 231-243.
Diwan, A., *et al*. "Compiler support for garbage collection in a statically typed language". ACM *PLDI'92, Sigplan Not. 27*,6 (June 1992), 273-282.
Edelson, D.R. "Smart pointers: They're smart but they're not pointers". *Proc. Usenix C++ Tech. Conf. 92*, 1-19.
Edelson, D.R. "Precompiling C++ for Garbage Collection". In [Bekkers92], 299-314.
Ellis, J.R., *et al*. "Real-time concurrent collection on stock multiprocessors". ACM *PLDI'88*.
Friedman, D.P., and Wise, D.S. "Aspects of applicative programming for parallel processing". *IEEE Trans. Comput. C-27*,4 (Apr. 1978), 289-296.
Gabriel, R.P. *Performance and Evaluation of Lisp Systems*. MIT Press, Camb., MA 1985.
Gelernter, H., *et al*. "A Fortran-compiled list processing language". *J. ACM 7* (1960), 87-101.
Girard, J.-Y. "Linear Logic". *Theoretical Computer Sci. 50* (1987),1-102.
Goldberg, B. "Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme". ACM *PLDI'89, Sigplan Not. 24*,7 (July 1989), 313-321.
Inamura, Y., et al. "Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2". *Logic Programming, Proc. North American Conf.*, MIT Press, 1989, 907-921.
Kieburtz, R.B. "Programming without pointer variables". *Proc. Conf. on Data: Abstraction, Definition and Structure, Sigplan Not. 11* (special issue 1976), 95-107.
Lafont, Y. "The Linear Abstract Machine". *Theor. Comp. Sci. 59* (1988), 157-180.
Lemaître, M., *et al*. "Mechanisms for Efficient Multiprocessor Combinator Reduction". *Lisp & Funct. Progr. 1986*,.
Nilsen, K. "Garbage collection of strings and linked data structures in real time". *SW—Prac. & Exper. 18*,7 (1988).
Nishida, K., *et al*. "Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures". *Logic Programming, Proc. 7th Intl. Conf.*, MIT Press, 1990, 83-95.
Park, Y.G., and Goldberg, B. "Reference Escape Analysis: Optimizing Reference Counting based on the Lifetime of References". *Proc. PEPM'91*, Yale Univ., June, 1991, 178-189.
Shalit, A. *Dylan™: An object-oriented dynamic language*. Apple Computer, Camb., MA, 1992.
Steele, G.L. "Fast Arithmetic in MacLisp". AI Memo 421, MIT AI Lab., Camb., MA, Sept. 1977.
Steele, G.L. *Common Lisp, The Language; 2nd Ed.* Digital Press, Bedford, MA, 1990.
Stoye, W.R., *et al*. "Some practical methods for rapid combinator reduction". *Lisp & Funct. Progr. Conf.* 1984.
Strom, R.E. "Mechanisms for Compile-Time Enforcement of Security". *Proc. ACM POPL 10*, Jan. 1983.
Strom, R.E., and Yemini, S. "Typestate: A Programming Language Concept for Enhancing Software Reliability". *IEEE Trans. SW Engrg. SE-12*,1 (Jan. 1986), 157-171.
Wadler, P. "Is there a use for linear logic?". *Proc. ACM PEPM'91*, New Haven, June 1991, 255-273.
Wakeling, D., and Runciman, C. "Linearity and Laziness". *Proc. Funct. Progr. & Computer Arch.*, LNCS 523, Springer-Verlag, Aug. 1991, 215-240.
Wise, D.S., and Friedman, D.P. "The one-bit reference count". *BIT 17*,3 (Sept. 1977), 351-359.
Wise, D.S. "Stop-and-copy and One-bit Reference Counting". TR-360, Indiana U., Bloomington, IN, Oct. 1992.
Yuasa, T. "Design and Implementation of Kyoto Common Lisp". *J. Info. Proc. 13*,3 (1990), 284-295.

APPENDIX I. A SHORT TUTORIAL ON "LINEAR" LISP

Linear Lisp is a style of Lisp in which every bound name is referenced exactly once. Thus, each parameter of a function is used just once, as is each name introduced via other binding constructs such as `let`, `let*`, etc. A linear language requires work from the programmer to make explicit any copying or deletion, but he is paid back by better error checking during compilation and better utilization of resources (time, space) at run-time. Unlike Pascal, Ada, C, and other languages providing explicit deletion, however, *a linear language cannot have dangling references*.

The `identity` function is already linear, but `five` must dispose of its argument before returning the value `5`:

```
(defun identity (x) x)
(defun five (x) (kill x) 5)          ; a true Linear Lisp would use "x" instead of "(kill x)"
```

The `kill` function, which returns *no* values, provides an appropriate "boundary condition" for the parameter `x`. The appearance of `kill` in `five` signifies *non-linearity*. (See below for a definition of `kill`).

The `square` function requires *two* occurrences of its argument, and is therefore also non-linear. A second copy can be obtained by use of the `dup` function, which accepts one argument and returns *two* values—i.e., two copies of its argument. (See below for a definition of `dup`). The `square` function follows:

```
(defun square (x)
  (let* ((x x-prime (dup x)))        ; Use Dylan-style syntax for multiple values [Shalit92].
    (* x x-prime)))
```

Conditional expressions such as `if`-expressions require a bit of sophistication. Since only one "arm" of the conditional will be executed, we relax the "one-occurrence" linearity condition to allow a reference in both arms.[5] One should immediately see that linearity implies that an occurrence in one arm *if and only if* there is an occurrence in the other arm. (This condition is similar to that for *typestates* [Strom83] [Strom86]).

The boolean expression part of an `if`-expression requires more sophistication. Strict linearity requires that any name used in the boolean part of an `if`-expression be counted as an occurrence. However, many predicates are "shallow", in that they examine only a small (i.e., shallow) portion of their arguments (e.g., `null`, `zerop`), and therefore a modified policy is required. We have not yet found the best syntax to solve this problem, but provisionally use several new `if`-like expressions: `if-atom`, `if-null`, `if-zerop`, etc. These `if`-like expressions require that the boolean part be a simple name, which does not count towards the "occur-once" linearity condition. This modified rule allows for a shallow condition to be tested, and then the name can be reused within the arms of the conditional.[6]

We require a mechanism to *linearly* extract both components of a Lisp cons cell, since a use of `(car x)` precludes the use of `(cdr x)`, and vice versa, due to the requirement for a single occurrence of `x`. We therefore introduce a "destructuring let" operation `dlet*`, which takes a series of binding pairs and a body, and binds the names in the binding pairs before executing the body. Each binding pair consists of a pattern and an expression; the expression is evaluated to a value, and the result is matched to the pattern, which consists of list structure with embedded names. The list structure must match to the value, and the names are then bound to the portions of the list structure as if the pattern had been *unified* with the value. Linearity requires that a name appear only once within a particular pattern. Linearity also requires that each name bound by a `dlet*` binding pair must occur either within an expression in a succeeding binding pair, or within the body of the `dlet*` itself. Using these constructs, we can now program the `append` and factorial (`fact`) functions:

```
(defun lappend (x y)                                  ; append for "linear" lists.
  (if-null x (progn (kill x) y)                                     ; trivial kill
    (dlet* (((carx . cdrx) x))                       ; disassociate top-level cons.
      (cons carx (lappend cdrx y)))))     ; this cons will be optimized to reuse input cell x.

(defun fact (n)
  (if-zerop n (progn (kill n) 1)                                    ; trivial kill.
    (let* ((n n-prime (dup n)))                  ; Dylan-style multiple-value syntax.
      (* n (fact (1- n-prime))))))
```

Below we show one way to program the `kill` and `dup` functions.

```
(defun kill (x)          ; Return no values.  Expensive way to decrement a reference count.
  (if-atom x (kill-atom x)
    (dlet* (((carx . cdrx) x))
      (kill carx) (kill cdrx))))

(defun dup (x)            ; Return 2 values.  Expensive way to increment a reference count.
  (if-atom x (dup-atom x)
    (dlet* (((carx . cdrx) x))
      (let* ((carx carx-prime (dup carx)) (cdrx cdrx-prime (dup cdrx)))
        (share-if-possible (cons carx cdrx) (cons carx-prime cdrx-prime))))))   ; reuse input.
```

---

[5]Any use of parallel or *speculative* execution of the arms of the conditional would require strict linearity, however.

[6]Although this rule seems a bit messy, it is equivalent to having the shallow predicate return *two* values: the predicate itself and the unmodified argument. This policy is completely consistent with linear semantics.

