ACM *Sigplan Notices* **29**,2 (Feb. 1994), 13-18.

A "Linear Logic" Quicksort

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.

The *linear* style of programming inspired by *linear logic* has been proposed to reduce garbage collection and synchronization costs in serial and parallel systems. We programmed Quicksort for both lists and arrays in a "linear" fragment of Lisp to estimate the performance impact of linearity on a serial machine. Even though Quicksort is well-tuned for current non-linear architectures, we find that linearity extracts no real penalty. Our "linear" list Quicksort is as fast as any non-linear list Quicksort, and our "linear" vector Quicksort is only 3.5% slower than a non-linear vector Quicksort. The linear style is moderately pleasant, and the redundancy of linearity checking can aid in finding bugs.

INTRODUCTION

Linear logic [Girard87] [Lafont88] [Abramsky93] has been proposed as the basis for a "linear" computer language which preserves the cleanliness of functional programming, yet allows efficient "update-in-place" array operations, no tracing garbage collection and no synchronization. However, some early measurements on linear languages have been disappointing. Wakeling [Wakeling91] complains about the inefficiencies of his version of "linear" ML, especially on list and array variants of Quicksort sorting algorithm, as well as about the stilted linear programming style.

We have programmed several versions of Quicksort in a linear fragment of Common Lisp, and—contrary to the conclusions of [Wakeling91]—find that Linear Lisp produces a very fast Quicksort routine. In fact, a linear Quicksort routine for lists (which doesn't require garbage collection) is considerably faster than a non-linear Quicksort routine for lists (which does require garbage collection), while the linear Quicksort routine for vectors is only 3.5% slower than the non-linear Quicksort routine. We do not find the linear style to be particularly burdensome for programming Quicksort, again disputing the conclusions of [Wakeling91], especially when arrays are subdivided instead of indexed.

A SHORT TUTORIAL ON "LINEAR" LISP

"Linear" Lisp is a style of Lisp in which each 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 the programmer to make explicit any copying and 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 never has dangling references.*

The `identity` function is 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 simply 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 Appendix for a definition of `kill`).

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

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

Conditional expressions—e.g., `if`-expressions—require sophistication. Since only one of the arms of the conditional is executed (assuming non-speculative computation), we can relax the linearity condition to allow an occurrence in both arms. Linearity then implies that there exists an occurrence in one arm *if and only if* there is also an occurrence in the other arm. (This condition is similar to that for *typestates* [Strom83]).

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 more efficient. We have not yet found the best syntax for shallow predicates, 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.[1]

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

We require a mechanism to *linearly* extract both components of a Lisp cons cell, since any 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 strictly evaluated to a value, and this value 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. As expected, linearity demands that a name appear only once within a particular pattern. Linearity furthermore requires that each name bound by a `dlet*` binding pair must occur exactly once—either within an expression in a succeeding binding pair, or within the body of the `dlet*`.

Using these constructs, we can program the familiar `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))
      (lcons carx (lappend cdrx y)))))              ; lcons will be optimized.
(defun fact (n)
  (if-zerop n (progn (kill n) 1)                               ; trivial kill.
    (let* ((n n-prime (dup n)))
      (* n (fact (1- n-prime))))))
```

NON-LINEAR QUICKSORT FOR LISTS

Quicksort [Knuth73] [Baase78] is a simple and efficient sorting algorithm for a set of elements that selects one test element and partitions the other elements into those which are smaller (<) than the test element and those which are larger (≥). These subsets are then themselves recursively sorted using the same algorithm. In Lisp/Scheme style, the partitioning loop is *tail-recursive*. Although the sorting of the subsets is inherently parallel, it is efficient on a serial architecture to sort the lower subset *onto* the sorted list of the upper subset to avoid appending the lists together.

```
(defun qs (x l)                         ; sort the list x onto the list l.
  (if (null x) l
    (let* ((i (car x)) (restx (cdr x))
           (high low (highlow restx i nil nil)))
      (qs low (cons i (qs high l))))))
(defun highlow (x i h l)     ; select the high and low elts of x onto h and l.
  (if (null x) (values h l)
    (let* ((firstx (car x)) (restx (cdr x)))
      (if (< firstx i) (highlow restx i h (cons firstx l))
        (highlow restx i (cons firstx h) l)))))
```

This applicative version of Quicksort is not only quite efficient, but almost in linear form.

LINEAR QUICKSORT FOR LISTS

To put Quicksort into linear form, we change the code until each bound variable occurrence is used exactly once. We use the `kill` function to dispose values, and the `dup` function to copy values. The additional `i` value returned by `lhighlow` is characteristic of programming in linear style. We have to do something with `i`, and since it is needed by its caller anyway, we return it. Although this effort is silly for a small integer (fixnum), it is quite appropriate in the case of a very large integer (bignum), record structure or an abstract data type with a specialized copying method.

```
(defun lqs (x l)
  (if-null x (progn (kill x) l)
    (dlet* (((i . restx) x))
      (let* ((high low i (lhighlow restx i nil nil)))
        (lqs low (lcons i (lqs high l)))))))
(defun lhighlow (x i h l)
  (if-null x (progn (kill x) (values h l i))
    (dlet* (((firstx . restx) x))
      (let* ((truth firstx i (l< firstx i)))
          (if truth
            (lhighlow restx i h (lcons firstx l))
            (lhighlow restx i (lcons firstx h) l)))))
```

This Quicksort routine is in linear form, and executes without producing any garbage, since `lcons` re-uses any list cells recycled by `dlet*`. A true Linear Lisp compiler would optimize further by deferring the memory cycles required to put the cells recycled by `dlet*` back onto the freelist until the scope of the `dlet*` expression is exited. Then if the compiler sees an `lcons` within this scope, it can use one of these recycled cells immediately without the cost of popping a cell from the freelist. For example, the cell recycled by the `dlet*` in `lqs` can be utilized in the later `lcons`. Similarly, the cell recycled by the `dlet*` in `lhighlow` can be utilized in both mutually exclusive arms of the conditional. Thus, in both these functions the linear compiler can avoid putting cells onto the freelist and taking them off the freelist. In other words, an optimized linear Quicksort *never* calls the real `cons` function! We can prototype this optimization by using a macro `lcons` whose third argument is such a recycled cell.

```
(defmacro lcons (a d c) `(rplacd (rplaca ,c ,a) ,d))
(defun lqs (x l)
  (if-null x (progn (kill x) l)
    (dlet* (((i . restx) x))
      (let* ((high low i (lhighlow restx i nil nil)))
        (lqs low (lcons i (lqs high l) x))))))
(defun lhighlow (x i h l)
  (if-null x (progn (kill x) (values h l i))
    (dlet* (((firstx . restx) x))
      (let* ((truth firstx i (l< firstx i)))
          (if truth
            (lhighlow restx i h (lcons firstx l x))
            (lhighlow restx i (lcons firstx h x) l))))))
```

To preserve linearity, we must change slightly the semantics of `dlet*`. If `dlet*` *swaps* [Baker92LLL] the `car` and the `cdr` of `x` into the pattern variables, `x` will be left bound to a cell with an empty `car` and `cdr`. The later call to `lcons` swaps the new `car` and `cdr` into `x` and returns it as the consuming occurrence of `x`. At no time is there ever more than one reference outstanding to the cell or any of its components, so linearity is preserved.

We tested this version of Quicksort on Coral Common Lisp 1.2 on a Macintosh Plus with a 16MHz Radius 68020 accelerator card. We sorted lists of random fixnums of various lengths, and it took approximately $45 \bullet n \bullet \log_2 n$ μsecs to sort a random list of n fixnums, or 12.73 secs. for 20,000 fixnums. This timing was 2.16X faster than the built-in `sort` function, although some of this difference can be attributed to the fact that the built-in `sort` does not know until run-time what function to use for comparison. Our linear Quicksort was also 1.84X faster than the built-in `sort` routine for a *vector* of 20,000 random fixnums.

We found that the Coral Common Lisp compiler did not do a good job of optimizing tail recursion. In fact, when we moved the `lhighlow` function inside the `lqs` function—normally an optimization—this sorting routine got much slower due to the creation of unnecessary closures. In order to estimate the speed with properly optimized tail recursion, we performed this optimization by hand by converting it into a Common Lisp `prog` form with labels and `go`'s. This version was 1.24X faster with a timing of about $36 \bullet n \bullet \log_2 n$ μsecs. With a generic comparison predicate, it then took $52.1 \bullet n \bullet \log_2 n$ μsecs, or 1.81X faster than the built-in generic `sort` routine for lists.

We thus have produced an extremely efficient linear Quicksort algorithm. In fact, we are not aware of any further optimizations possible, outside of better register optimization, or perhaps more in-lining (loop unrolling). Thus, our linear Quicksort is not only competitive with the best non-linear algorithm, it *is* the best *non*-linear algorithm.

NON-LINEAR QUICKSORT FOR VECTORS

Quicksort for vectors is somewhat less elegant than for lists when programmed in Lisp. We use a tail-recursive style for the inner "two-pointer" split loop [Baase78].

```
(defun vqs (v k m)                          ; Quicksort vector v from k up to m.
  (if (>= k m) v
    (let* ((x (aref v k))                                   ; Create a hole at k.
           (i (split1 v k (1- m) x)))                          ; Do partition.
      (setf (aref v i) x)                             ; Put x back into the hole.
      (vqs v k i)                                   ; Quicksort v from k up to i.
      (vqs v (1+ i) m))))                         ; Quicksort v from i+1 up to m.
(defun split1 (v i j x)                                       ; hole is at v(i).
  (if (= i j) i
    (let* ((vj (aref v j)))                                   ; Copy elt. to vj.
      (if (< vj x)
        (progn (setf (aref v i) vj) (split2 v (1+ i) j x))
        (split1 v i (1- j) x)))))
(defun split2 (v i j x)                                       ; hole is at v(j).
  (if (= i j) i
    (let* ((vi (aref v i)))                                   ; Copy elt. to vi.
      (if (>= vi x)
        (progn (setf (aref v j) vi) (split1 v i (1- j) x))
        (split2 v (1+ i) j x)))))
```


LINEAR QUICKSORT FOR VECTORS

Converting a non-linear vector Quicksort into linear style requires several changes in order to produce an efficient algorithm. First, the vector must be consistently passed as an argument and returned as a value. Second, every vector reference becomes a *swap*, which installs a new value while returning the old value together with the vector itself and the index. Third, every bound name must occur exactly once, including those for vector indices, so multiple uses require explicit copies. Fourth, we require certain predicates to additionally return their arguments unchanged.

Swapping instead of *copying* elements from a vector can introduce an inefficiency into linear Quicksort unless the partitioning loop is tuned to take advantage of this behavior. In a nonswapping Quicksort, each vector element is implicitly copied to a register and then compared to the test element. If the vector element is already in its proper partition, it does not have to be restored. However, if not, it is stored into the hole created by a previous movement. In a linear Quicksort, each vector element is swapped into a register and then compared to the test element. It is therefore better to have the register hold an element value before the swap, so that the swap can simultaneously perform both reading and writing. We can "prime the pump" by having the register initialized with an element that should be stored in the location of the next swap. Thus, the circuitry transferring data in both directions is utilized.

```
(defun lvqs (v k m)                  ; Quicksort vector v from k up through m.
  (let* ((truth k m (l>= k m)))
    (if truth (progn (kill k) (kill m) v)
      (let* ((x v k (laref v k 'hole))                      ; Get test element.
             (k1 (1+ k))
             (ve v k1 (laref v k1 'hole))                  ; Get lower element.
             (ve x (sort2 ve x))  ; Conditionally exchange ve, x so that ve≤x.
             (k1 k1-prime (dup k1))                              ; Trivial dup.
             (m m-prime (dup m))                                 ; Trivial dup.
             (v i ve x (lsplitlow v k1-prime m-prime ve x))    ; Do partition.
             (ve v k1 (laref v k1 ve))              ; Put back lower element.
             (x v i (laref v i x))                     ; Put in test element.
             (k (1- k1))
             (x v k (laref v k x))                  ; Put back lower element.
             (i i-prime (dup i)))                                ; Trivial dup.
        (lvqs (lvqs v k (1- i-prime)) (1+ i) m)))))
(defun lsplitlow (v i j ve x)                                  ; ve≤x, i≤j.
  (let* ((truth i j (l= i j)))
    (if truth (progn (kill j) (values v i ve x))
      (let* ((i (1+ i))
             (ve v i (laref v i ve))       ; Swap lower elt. with unknown elt.
             (truth ve x (l<= ve x)))
        (if truth (lsplitlow v i j ve x)
          (lsplithigh v i j ve x))))))
(defun lsplithigh (v i j ve x)                                 ; ve>x, i≤j.
  (let* ((truth i j (l= i j)))
    (if truth
      (let* ((ve v i (laref v i ve)))        ; Swap upper elt. with lower elt.
        (kill j) (values v (1- i) ve x))
      (let* ((ve v j (laref v j ve))       ; Swap upper elt. with unknown elt.
             (j (1- j))
             (truth ve x (l<= ve x)))
        (if truth (lsplitlow v i j ve x)
          (lsplithigh i j ve x))))))
```

The requirement for continually passing around the vector is not onerous. However, the requirement to explicitly copy the index values is pedantic. A type system allowing both linear and non-linear types, with the non-linear types used for things like array indices, would allow for a more pleasant style for this task. Even without such types, however, a compiler should find it easy to optimize trivial duplications.

As in the list Quicksort case, our vector Quicksort was slow due to poor tail-recursion optimization by the Coral Common Lisp compiler. We performed appropriate tail-recursion optimizations by hand for both the non-linear and linear versions of our vector Quicksort, and got substantial speedups. Since Coral Common Lisp does not handle the returning of multiple values efficiently, we performed additional hand optimizations to understand their value in this context. We found that the efficient handling of multiple values is critical to achieving acceptable performance on our linear benchmark. In particular, we assumed that built-in linear versions of predicates which returned multiple values could be compiled as efficiently as in a normal applicative system where the additional values did not need to be returned. We also assumed that the built-in array reference swapping instructions would be similarly optimized.

The final linear generic[2] swapping Quicksort is only 3.5% slower than the non-linear generic non-swapping Quicksort. We could achieve parity if we could eliminate the redundant index calculations which result from the emulation of swapping on an assignment-oriented architecture. Furthermore, if a hardware swap operation takes the same amount of time as a hardware read operation (we have every reason to believe it can), then a swapping Quicksort can be strictly faster than a non-linear Quicksort.

A MORE ELEGANT LINEAR QUICKSORT FOR VECTORS

The vector linear Quicksort routine above doesn't take full advantage of the linearity of the vector. Unlike nonlinear vectors, a linear vector can be linearly partitioned in-place to yield non-overlapping subvectors. These subvectors can then be linearly concatenated back to form the larger vector, also in-place, so long as the concatenation is contiguous. A linear Quicksort should be inspired by the notion that linear objects are *resources*, and manipulate the pieces of the vector itself, rather than indices. Vector decomposition/concatenation can thus be used for a very elegant Quicksort.

```
(defun lvqs (v)                                     ; Quicksort the linear vector v in-place.
  (if-empty v v                                                  ; If v is empty, we are done.
    (let* ((hole v (first&rest v))       ; Split vector into subvectors of lengths 1 and n-1.
           (x hole zero (laref hole 0 'empty))                   ; Get test element from hole.
           (x lo hole hi (part1 x 0 'empty '#() hole v '#()))              ; Partition vector.
           (hv hole zero (laref hole zero x))               ; Put test element back into hole.
           (lo (lvqs lo))                                               ; Sort low partition.
           (hi (lvqs hi)))                                             ; Sort high partition.
      (kill zero) (kill hv)                                                  ; Trivial kills.
      (catenate lo hole hi))))                                ; Return reconstructed vector.

(defun part1 (x zero hv lo hole1 mid hi)        ; Partition mid using x; v = lo:hole1:mid:hi.
  (if-empty mid (progn (kill zero) (kill hv) (kill mid) (values x lo hole1 hi))
    (let* ((mid hole2 (rest&last mid))                 ; Split off last element from mid.
           (y hole2 zero (laref hole2 zero hv))            ; Extract element y from hole2.
           (truth y x (l< y x)))                           ; Compare y with test element.
      (if truth
        (let* ((hv hole1 zero (laref hole1 zero y)))                     ; Put y into hole1.
          (part2 x zero hv (catenate lo hole1) mid hole2 hi))       ; Reuse lo,hole1 storage.
        (let* ((hv hole2 zero (laref hole2 zero y)))                     ; Put y into hole2.
          (part1 x zero hv lo hole1 mid (catenate hole2 hi)))))))    ; Reuse hole2,hi storage.

(defun part2 (x zero hv lo mid hole2 hi)        ; Partition mid using x; v = lo:mid:hole2:hi.
  (if-empty mid (progn (kill zero) (kill hv) (kill mid) (values x lo hole2 hi))
    (let* ((hole1 mid (first&rest mid))               ; Split off first element from mid.
           (y hole1 zero (laref hole1 zero hv))            ; Extract element y from hole1.
           (truth y x (l< y x)))                           ; Compare y with test element.
      (if truth
        (let* ((hv hole1 zero (laref hole1 zero y)))                     ; Put y into hole1.
          (part2 x zero hv (catenate lo hole1) mid hole2 hi))       ; Reuse lo,hole1 storage.
        (let* ((hv hole2 zero (laref hole2 zero y)))                     ; Put y into hole2.
          (part1 x zero hv lo hole1 mid (catenate hole2 hi)))))))    ; Reuse hole2,hi storage.
```

We tested this routine, but could not prototype an optimized version because Common Lisp does not trust the user to destroy and compose vectors in this fashion. However, we believe that this style of array manipulations can be quite efficient if linear vectors are implemented by means of separate headers. The recursions would manipulate only the headers, which by linearity could always be located in registers. With proper tail recursion optimizations, the number of parameters in the tail recursions would not affect performance, and the overall performance should be good. By additionally arranging swaps to utilize communication in both directions, as before, performance should be excellent.

A vector subdivision Quicksort is ideal for a parallel processing machine—whether using parallel functional units (VLIW) or parallel instruction streams—because the vector is subdivided into exclusively-owned partitions. These subvectors can then be sorted completely independently and in parallel. The linear style, in which all events are ordered solely by dataflow constraints, makes trivial the compiler's job of detecting interference. The linear style recognizes that every access to a register—e.g., a read used to perform a comparison—ties up the register and therefore requires that the register be passed as an argument and returned as a value to guarantee proper ordering. Yet the linearity constraint also allows the factoring of the binding environment so that subexpressions can easily be reordered or executed in parallel. Quicksort may not be the ideal parallel sorting routine, but this linear subdividing Quicksort automatically extracts as much parallelism as possible.

---

[2]The comparison predicate passed to a linear generic Quicksort returns 3 values: a boolean value and the two unmodified arguments. Such a predicate allows the linear Quicksort to avoid *all* copying of array element values.

CONCLUSIONS

A linear programming style can be as efficient as a non-linear style, and still be pleasant to write and read. However, the kinds of compiler optimizations required for efficient linear-style programs are different from those stressed by non-linear styles of programming. In particular, the efficient handling of multiple values, array element swaps, and operations on trivial data types—e.g., small integers—become important. A tuning of the Quicksort algorithm to take advantage of array element swapping was found to bring the performance of the "linear" vector Quicksort to within 3.5% of the non-linear vector Quicksort, where both were implemented on the same 68020 architecture.

Many may find onerous the linearity requirement that all names be explicitly consumed. However, we have found that this linearity check is an excellent programming aid, because—for example—it makes sure that every parameter of a function is either used or given a reason why not.

Linearity in a language is closely related to the *typestates* [Strom83] of NIL/Hermes, and to *islands* [Hogg91].

REFERENCES

Abramsky, S. "Computational interpretations of linear logic". *Theor. Comp. Sci. 111* (1993), 3-57.

Aho, A.V., *et al*. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

Baase, S. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, MA, 1978.

Baker, H.G. "The Buried Binding and Dead Binding Problems of Lisp 1.5: Sources of Incomparability in Garbage Collector Measurements". ACM *Lisp Pointers V*,2 (Apr.-June 1992), 11-19.

Baker, H.G. "Lively Linear Lisp — 'Look Ma, No Garbage!'". ACM *Sigplan Notices 27*,8 (Aug. 1992), 89-98.

Baker, H.G. "NREVERSAL of Fortune—The Thermodynamics of Garbage Collection". *Int'l. W/S on Memory Mgmt*., St Malo, France, Sept. 1992, Springer LNCS 637.

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.

Friedman, D.P., and Wise, D.S. "Aspects of applicative programming for parallel processing". *IEEE Trans. Comput. C-27*,4 (Apr. 1978), 289-296.

Girard, J.-Y. "Linear Logic". *Theoretical Computer Sci. 50* (1987),1-102.

Harms, D.E., and Weide, B.W. "Copying and Swapping: Influences on the Design of Reusable Software Components". *IEEE Trans. SW Eng. 17*,5 (May 1991),424-435.

Hesselink, W.H. "Axioms and Models of Linear Logic". *Formal Aspects of Comput. 2*,2 (Apr-June 1990), 139-166.

Hogg, J. "Islands: Aliasing Protection in Object-Oriented Languages". *Proc. OOPSLA'91, Sigplan Not. 26*,11 (Nov. 1991), 271-285.

Knuth, D.E. *The Art of Computer Programming, v. 3, Sorting and Searching*. Addison-Wesley, Reading, MA 1973.

Lafont, Y. "The Linear Abstract Machine". *Theor. Comp. Sci. 59* (1988), 157-180.

Martí-Oliet, N., and Meseguer, J. "From Petri nets to linear logic". *Math. Struct. in Comp. Sci. 1*,1 (Mar. 1991).

Mendelson, A. "A Single Cached Copy Data Coherence Scheme for Multiprocessor Systems". *Comput. Arch. News 17*,6 (Dec. 1989), 36-49.

Shalit, A. *Dylan™: An object-oriented dynamic language*. Apple Computer, Camb., MA, 1992.

Strom, R.E. "Mechanisms for Compile-Time Enforcement of Security". *Proc. 10th ACM POPL*, Jan. 1983.

Suzuki, N. "Analysis of Pointer 'Rotation'". *CACM 25*,5 (May 1982)330-335.

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.

APPENDIX

```
(defun kill (x)
  ;;; Return no values.
  (if-atom x (kill-atom x)
    (dlet* (((carx . cdrx) x))
      (kill carx) (kill cdrx)
      (values))))
(defun dup (x)
  ;;; Return 2 values.
  (if-atom x (dup-atom x)
    (dlet* (((carx . cdrx) x))
      (let* ((carx carx-prime (dup carx)) (cdrx cdrx-prime (dup cdrx)))
        (values (lcons carx cdrx) (lcons carx-prime cdrx-prime))))))
```

