ACM *Sigplan Notices* **27**,8 (Aug. 1992),89-98.

Lively Linear Lisp — 'Look Ma, No Garbage!'[1]

Henry G. Baker

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

Abstract

Linear logic has been proposed as one solution to the problem of garbage collection and providing efficient "update-in-place" capabilities within a more functional language. Linear logic conserves accessibility, and hence provides a *mechanical metaphor* which is more appropriate for a distributed-memory parallel processor in which copying is explicit. However, linear logic's lack of sharing may introduce significant inefficiencies of its own.

We show an efficient implementation of linear logic called *Linear Lisp* that runs within a constant factor of non-linear logic. This Linear Lisp allows RPLACX operations, and manages storage as safely as a non-linear Lisp, but does not need a garbage collector. Since it offers assignments but no sharing, it occupies a twilight zone between functional languages and imperative languages. Our Linear Lisp Machine offers many of the same capabilities as combinator/graph reduction machines, but without their copying and garbage collection problems.

Introduction

> **Neither a borrower nor a lender be;**
> **For loan oft loses both itself and friend ...** [Shakespeare, *Hamlet*, I. iii 75]

The sharing of data structures can be efficient, because sharing can substitute for copying, but it creates ambiguity as to who is responsible for their management. This ambiguity is difficult to statically remove [Barth77] [Bloss89] [Chase87] [Hederman88] [Inoue88] [Jones89] [Ruggieri88], and we have shown [Baker90] that static sharing analysis—even for pure functional languages—may be as hard as ML type inference, which is known to be exponential [Mairson90].[2] We show here that a system which takes advantage of *incidental*, rather than *necessary*, sharing can provide efficiency without the ambiguity normally associated with sharing.

Linear logic was invented by Girard [Girard87] as a way to deal with objects which should not be blithely shared or copied. Wadler [Wadler91] has proposed the use of data structures with unity reference counts in order to avoid the problems of garbage collection and provide for O(1) array update. Wakeling [Wakeling91] has implemented linear logic and found that it is extremely slow, due to its requirement for laborious copying. Our Linear Lisp Machine implements unity reference count data structures more efficiently than Wakeling's machine, and should be "competitive" with traditional implementations of combinator/graph reduction machines.

A Linear Lisp Machine

In this section we introduce an automata-theoretic model of a Lisp Machine in which all cons cells have reference counts of exactly 1, which implies that all data structures are *trees*—i.e., they have no sharing or cycles. For example, in our Linear Lisp, `NREVERSE` is isomorphic to `REVERSE`.

Our machine consists of a finite state control and n pointer registers which can hold either *atoms* or pointers to *cons* cells. An *atom* is either `NIL` or a *symbol*. One of the registers—`fr`—is distinguished as the "free list" register, and is initialized to point to an infinite list of `NIL`'s. Another register—`sp`—is designated as the "stack pointer" register.

The machine can execute any of the following atomic operations:

```
r1<->r2;           /* swap r1,r2. */
r1<->CAR(r2);      /* r1,r2 distinct; precondition(not ATOM(r2)) */
r1<->CDR(r2);      /* r1,r2 distinct; precondition(not ATOM(r2)) */
NULL(r1);          /* predicate for r1=NIL */
ATOM(r1);          /* predicate for r1=NIL or symbolp(r1) */
EQ(r1,r2);         /* defined only for atomic r1,r2; see [Baker93ER] */
r1:='foo;          /* precondition(ATOM(r1) and ATOM('foo)) constant 'foo */
r1:=r2;            /* precondition(ATOM(r1) and ATOM(r2)) */

CONS(r1,r2):       /* r1,r2 distinct; r2<-CONS(r1,r2) and set r1=NIL */
{r1<->CAR(fr); r2<->fr; fr<->CDR(r2);};
```

---

[1]Obscure reference to old Proctor & Gamble television advertisement for *Crest* toothpaste.

[2]We have conjectured that sharing analysis [Baker90] utilizing ML type inference will elicit its exponential behavior.


```
PUSH(r1,r2):      /* r1,r2 distinct; Push r1 onto r2 and set r1=NIL */
CONS(r1,r2);

POP(r1,r2):       /* r1,r2 distinct; Pop CAR(r2) into r1 if r1=NIL */
if NULL(r1) and not ATOM(r2)
   then {fr<->CDR(r2); r2<->fr; r1<->CAR(fr);}
   else die;
```

<u>Proposition 1</u>. List cell reference counts are conserved and are always identically 1.

Proof by induction [Suzuki82,s.4]. All cons cells start out with unity reference counts, and are only manipulated by exchanges which preserve reference counts. QED

<u>Proposition 2</u>. All list cells are always accessible—i.e. *live*, and no garbage is created.

Proof by induction [Suzuki82,s.5]. Storage could "leak" only if we performed r*i*<->CXR(r*i*), but we do not allow the swapping of a register with a portion of its own contents. QED

Programming Note: The only way to "clear" a register which points to a list is to decompose the list stored there and put it back onto the free list.[3]

```
FREE(r1):                              /* Essentially the K combinator! */
if not NULL(r1) then
   if ATOM(r1) then r1:='NIL;
   else
    {PUSH(r2,sp); POP(r2,r1);          /* temporary r2≠r1. */
     FREE(r1);4                        /* free the cdr of the cell. */
     r2<->r1; FREE(r1);                /* free the car of the cell. */
     POP(r2,sp);}
```

We can copy, but only by destroying the original list and creating two new lists.

```
COPY(r1,r2):            /* assert(r2=NIL).  Essentially the S combinator! */
if not NULL(r1) then
   if ATOM(r1) then r2:=r1;
   else
    {PUSH(t1,sp); PUSH(t2,sp);
     POP(t1,r1); COPY(r1,r2);
     t1<->r1; t2<->r2; COPY(r1,r2);
     t1<->r1; t2<->r2; PUSH(t1,r1); PUSH(t2,r2);
     POP(t2,sp); POP(t1,sp);}
```

Finally, we can program recursive EQUAL by destroying and recreating both lists. (We switch to Lisp notation in order to utilize the capabilities of prog1.)

```
EQUAL(r1,r2): /* Recursive list equality. */
(or (and (ATOM r1) (ATOM r2) (EQ r1 r2))
    (and (not (ATOM r1)) (not (ATOM r2))
         (progn (PUSH t1 sp) (PUSH t2 sp) (POP t1 r1) (POP t2 r2)
          (prog1
           (and (EQUAL r1 r2)
                (progn (<-> t1 r1) (<-> t2 r2)
                 (prog1 (EQUAL r1 r2)
                  (<-> t1 r1) (<-> t2 r2))))
           (PUSH t1 r1) (PUSH t2 r2) (POP t2 sp) (POP t1 sp)))))
```

Using these definitions, it should be clear that we can program a traditional Lisp interpreter (see Appendix). However, this interpreter will be inefficient, due to the extra expense of copying. The one minor problem to be faced

---

[3]One could use the Weizenbaum lazy recycling trick [Weizenbaum63] and put garbage onto the free list; this trick has also been advocated by [Wise85]. Putting garbage onto the free list moves work from the point of freeing to the point of allocation; this may smooth out the work, but does not decrease its amount, unless the program terminates with a dirty free list.

[4]There are implicit "old-PC" stack operations involved with recursive calls, but we leave those details to the reader.

is how to handle recursive functions, since we cannot create cycles. We suggest the following trick based on the lambda calculus Y combinator [Gabriel88]:

```
(defun fact (f n) (if (zerop n) 1 (* n (funcall f f (1- n)))))
(defun factorial (n) (fact #'fact n))
```

A Linear Lisp Machine with `FREE`, `COPY`, `EQUAL` and Assignment

In the previous section, we described a Linear Lisp Machine in which `FREE`, `COPY` and `EQUAL` had to be programmed. Now that we have seen how to implement these three functions, we will describe a new machine in which they are primitive operations—e.g., they are implemented in microcode. This revision will not provide much additional efficiency, but it will provide a more traditional set of primitive operations.

```
r1:=r2:                          /* r1, r2 cannot be the same register. */
{FREE(r1); COPY(r2,r1);}

r1:=CAR(r2):                     /* r1, r2 cannot be the same register. */
{r3<->CAR(r2); r1:=r3; r3<->CAR(r2);}

r1:=CDR(r2):                     /* r1, r2 cannot be the same register. */
{r3<->CDR(r2); r1:=r3; r3<->CDR(r2);}

RPLACA(r1,r2):       /* CAR(r1):=r2.  r1, r2 cannot be the same register. */
{r3<->CAR(r1); r3:=r2; r3<->CAR(r1);}

RPLACD(r1,r2):       /* CDR(r1):=r2.  r1, r2 cannot be the same register. */
{r3<->CDR(r1); r3:=r2; r3<->CDR(r1);}
```

Using this new set of operations, we can now program our Lisp interpreter in the traditional way, although this interpreter will now be slower unless we have taken precautions to avoid extraneous copying. This interpreter is still *linear* logic, in which there is no sharing, however, so operations like `RPLACX` cannot create loops.

Dataflow-like Producer/Consumer `EVAL`

Before showing how to make `FREE`, `COPY` and `EQUAL` more efficient, it is instructive to show a natural programming metaphor for Linear Lisp. The natural metaphor for Linear Lisp is quantum mechanics, in which objects interact, and every interaction with an object—including reading—affects the object.

Linear Lisp calls for the mechanical interpretation of a function as a "black box" which *consumes* its arguments and *produces* its result, while returning the black box to its quiescent state. Since an argument to a function is consumed, the function can reference it only once, after which the formal parameter binding becomes unbound! In fact, the requirement to return the box to its quiescent state means that every parameter and local variable must be referenced *exactly once*, since it is a run-time error to return from the function so long as any parameters or local variables still have bindings.[5] In order to utilize a value more than once, it must be explicitly copied. We relax the "reference-once" requirement in the case of multiple-arm conditionals. Since the execution of one arm implies the non-execution of the other arms, the single use of the variable within each arm is sufficient to guarantee that single use is dynamically satisfied.[6] In fact, the best policy is to reference exactly the same set of variables in all of the arms of the conditional. The exactly-one-reference policy can be checked syntactically at compile-time in a manner analogous to the "occurs free" predicate of the lambda calculus.

The following technique should make the programming of certain predicates more efficient. A value passed to a predicate is often subsequently used in the arm of a conditional, yet in many of these cases, the predicate does not depend upon any deep property of the value. The cost of copying and then consuming the *entire* value is therefore wasted. For such a "shallow" predicate, one might rather program it to return (a list of) two values—the value of the predicate and its reconstituted arguments, which is then bound to new parameters and (re)used in the subsequent computation.

Since the argument to `CAR` or `CDR` is completely consumed, how can one gain access to both components? The natural model for binding in Linear Lisp is that of a *destructuring bind*, which binds a number of variables "simultaneously", while recycling the backbone of the value-containing list structure. This destructuring bind eliminates the need for separate `CAR` and `CDR` functions.

[5] I.e., no values can be "left on base at the end of an inning", to provide a baseball analogy. This error can usually be checked at compile-time.

[6] Copying is still required if speculative execution of an arm is performed.

Nested functional composition has the obvious mechanical interpretation, since the intermediate results are utilized by exactly one consumer. The mechanical metaphor also shows that parallel execution of subexpressions is possible and correct (so long as the primitive `CONS` itself—the creator of argument *lists*—evaluates its arguments in parallel), since there is no mechanism whereby the subexpressions can communicate. Thus, collateral argument evaluation [Baker77] can be considered the norm, and on this machine there are no garbage collection problems because every callee is required to "clean up" after itself. Unfortunately, the hash consing technique described later requires that `CONS` be (transitively) strict in both of its arguments. As a result, only variable binding itself can be lazy, which isn't nearly lazy enough for most interesting applications of laziness; lazy "futures" [Baker77] cannot be supported.

The analogy with a dataflow machine [Arvind87] is quite close. Values are *tokens* which are consumed by a function to produce a new token/value. The implementation of the token metaphor on our Swapping Lisp Machine is straightforward. In addition to programmer-visible values, we also have "holes", which are the bindings of variables when they are "unbound". Argument passing is accomplished by "swap-in, swap-out" (reminiscent of [Harms91]), and "holes" flow backwards relative to values. When implemented in this way (see Appendix), `EVAL` avoids most copying, and should be reasonably efficient even without the hash consing scheme discussed in the next section.

Reconstituting Trees from Fresh Frozen Concentrate

We will now show how to implement a linear machine with the instructions `CONS`, `CAR`, `CDR`, `COPY`, `EQUAL`, `RPLACA`, `RPLACD`, `NULL`, `ATOM` and `EQ` in such a way that all of these instructions operate in O(1) average time. In other words, our machine will be as fast as a machine based on "hash consing" [Goto74], except that our machine can also execute `RPLACX`.

Our machine will operate on a "virtual heap", and the real heap will be separated from the CPU by a "read barrier" [Moon84]. More precisely, the cons cells pointed at directly by the machine registers will operate in the fashion described above, but any cons cells which are not directly pointed at by machine registers may be represented differently. In particular, any cons cells which are not directly pointed at by machine registers are represented in a hidden hash table which is built in such a way that list structures which are `EQUAL` will be represented by exactly the same cell in this hash table. This "hash cons" table is built inductively from cells having atomic `CAR`'s and `CDR`'s by hashing the addresses of non-atomic `CAR`'s and `CDR`'s; this scheme is called "hash consing" and under the appropriate circumstances the cost of a "hash cons" operation averages O(1). Furthermore, these cells can be dereferenced for `CAR` or `CDR` in O(1) time. These cells cannot be side-effected, however, so that `RPLACX` cannot be used directly on these hash-consed cells. We will manage this hash table using reference counts, since we intend that cells stored in this table are capable of being shared.

We now analyze the operation of the read barrier. If the CPU attempts to read the `CAR` or `CDR` of a cell referenced directly by a machine register, then the read barrier causes the cell in the hash table to be copied ("unshared") into a normal cell, and the reference count of the cell in the hash table is decremented. Similarly, if the CPU attempts to write the `CAR` or `CDR` of a cell referenced directly by a machine register, then the cell is copied into the hash table by "hash consing" (including incrementing the reference counter) and the copied pointer is stored into the `CAR`/`CDR` instead.

Since there are never more than *n* "normal" cells which can be seen by the CPU, because there are only *n* registers, and because none of these normal cells can be shared, we can eliminate the expense of allocating and deallocating these cells, and associate one of these normal cells with each machine register. Thus, all storage management effort is concentrated in the hash cons table.

Another optimization is that of keeping a "hidden" pointer in each normal cell to the entry in the hash table from which it was copied; if it is newly consed, then this pointer is empty. This pointer is used as a "hint" when hash consing, so that no searching will be necessary to share the cell in the hash consed heap. Of course, any side-effects to this normal cell will cause its "hint" pointer to be cleared, since we will now require a hash lookup in order to share the cell in the hash consed heap.

Since the free list seen by the CPU is an infinite list of `NIL`'s, we can represent it as a special *circular* list of one cell in the hash table whose reference count is not modified. In this case, the free list register is identical in nature to the other registers in the CPU; the only difference is in the hash table pointer stored in the `CDR` of the first cell on the list.

Of course, there is a real "free list" which is hidden from the CPU which is used to provide cells for the hash table when a never-before-hashed `<CAR,CDR>` configuration is seen. This "free list" is refreshed whenever the reference count of a table entry drops to zero. Depending upon our time/space tradeoff, we can either recycle hash table cells aggressively or lazily. If we recycle aggressively, then we may have to recycle an unbounded number of nodes at one time, which can create an unbounded delay. Alternatively, we can recycle lazily, in which case the `CAR` and `CDR` of a recycled cell are only marked as deleted, but not actually reclaimed, at the time that the reference count for the cell drops to zero. Notice, though, that any sublists of this "garbage" list are still hashable by the hash table and can become non-garbage at any time. Thus, the garbage is not really garbage, although it can only be used for a very special purpose until recycled for general use.

We can now describe the operation of our "fast" machine. All primitive operations, with the exception of `FREE`, `COPY` and `EQUAL`, operate exactly as if the CPU were operating on a "real heap" rather than a "virtual heap". `COPY` copies only the top-level cons cell, and "copies" the sublists by simply incrementing the reference counts of the `CAR` and `CDR` cells in the hash table. `EQUAL` compares only the top-level cons cell, and simply compares the addresses of the `CAR`'s and `CDR`'s for their locations in the hash table. Thus, `COPY` and `EQUAL` are both O(1) operations. If we utilize lazy recycling for hash table entries, then `FREE` is also an O(1) operation.

Linear Lisp `EVAL`

It is interesting to analyze the operation of a traditional Lisp interpreter written for this Linear Lisp Machine (see Appendix). Since traditional Lisp utilizes "applicative order" evaluation, an argument used multiple times is only evaluated once. The traditional problem in "call-by-need" evaluation has been the shared flag variable which indicates that the argument has already been computed; Linear Lisp utilizes its arguments exactly once, with an explicit copy required for additional uses, so the applicative-order/normal-order issue is moot and the shared flag variable is not necessary. Our use of hash consing requires transitive strictness on both of its arguments; otherwise `EQUAL`'ity could be decided prior to the determination of a lazy value, since `EQUAL` and `EQ` are equivalent for hash conses. Thus, parallel evaluation of arguments can be supported, but not laziness. On the other hand, the implicit synchronization required to strictly resolve a "future" [Baker77] can be efficiently performed with a swapping operation [Herlihy91].

Since the association list of variable bindings must be destroyed in order to search it anyway, the "shallow binding" technique utilizing "rerooting" [Baker78] is essentially optimal.

We *can* destroy the Lisp program during evaluation in the manner of combinator/graph reduction [Turner79] [Kieburtz85] [Kieburtz87] [Johnsson85] [Johnsson91]; indeed, we *must* destroy it, since we cannot reference it otherwise, as all of its reference counts are one!

Implications for Real Multiprocessors

Linear Lisp can be used in a "shared-heap-memory" multiprocessing configuration, in which the memory to be shared is actually the hash consed heap. Each CPU operates independently, with its "normal nodes" acting like a local cache into the heap. The read barrier logic is the "cache consistency protocol" for this hashed memory, which is simple because there is no visible sharing among CPU's. This parallel configuration can be used, e.g., for the collateral evaluation of arguments.

Hardware swapping has been advocated as a synchronization mechanism [Herlihy91]. However, even without hardware swaps, modern RISC compilers can optimize register-register swaps, while write-back caches reduce the cost of register-memory swaps; swaps should thus be relatively cheap even on a traditional load/store architecture.

Conclusions and Previous Work

Some have suggested that garbage collection not be done at all [White80] or after the program has finished [Moon84] (comment on the Boyer benchmark). Linear Lisp provides a hyper-clean environment in which garbage is never produced, and therefore garbage collection is not necessary.

Hash consing was invented by Ershov for the detection of common subexpressions in a compiler [Ershov58] and popularized by Goto for use in a Lisp-based symbolic algebra systems [Goto74] [Goto76] [Goto80]. While a hash-consing system with reference count management can be used to implement a functional (applicative) subset of Lisp with the same efficiency shown here, we believe that our Linear Lisp Machine is the first efficient implementation which allows for (linear) side-effects such as `RPLACX`. See [Baker92] for a "warp speed" implementation of the Gabriel "Boyer" benchmark using hash consing.

Linear Lisp is an ideal environment for symbolic algebra, since it provides the efficiencies of sharing, including fast copying and fast equality checking [Goto76], without the problems. For example, the Macsyma symbolic algebra system can represent the symbolic determinant of an nxn matrix with $O(n^3)$ cons cells, even though this expression prints out with O(n!) terms. Furthermore, Linear Lisp allows for destructive operations on expressions, which can sometimes be more efficient [Gabriel85] [Fateman91], yet these destructive operations are completely safe.

While our Linear Lisp cannot support laziness, because `CONS` is transitively strict in both arguments, it does support a mild form of side-effects. Linear Lisp `RPLACX` cannot be used to produce (visible) sharing, and hence requirements for "object identity" expressed in [Baker93ER] are vacuously met for cons cells. These cells live in a twilight zone between functional and non-functional data; any "side-effects" to the data are not really "side"-effects because they are not visible through any other pointer alias. [Myers84] describes a scheme for implementing certain imperative data structures efficiently using applicative data structures. We believe that the implementation of side-effects in Linear Lisp *automatically* produces the efficiency claimed by his scheme, without translating the program into applicative form.

[Harms91] discusses the advantages of unity reference count data structures and swapping for efficient programming of abstract data types, although he utilizes notions as "unshared" or "non-aliased" instead of unity reference counts. [Kieburtz76] also advocates the use of hidden (unity reference count) pointers. Pointer swapping can be used to minimize reference count overflows in systems with limited counts [Wise77], and to avoid appearing multiply referenced [Deutsch76]. Memory-to-memory swapping is a superior form of synchronization [Herlihy91], so we expect to see efficient swapping operations implemented on future shared-memory multiprocessors.

Our Linear Lisp Machine *consumes* the programs it interprets, therefore requiring a private copy of the code in the manner of a combinator reduction machine. The Linear Lisp `COPY` is more efficient, however, than the real copying utilized in combinator reduction machines. A machine which consumes its programs provides new insight into the mechanisms of instruction caches (see also [Kieburtz87]) and "index registers". Since index registers were invented in order to avoid side-effecting code, and since all modern CPU's utilize instruction caches, the index register is obsolete! Linear logic provides a firm semantics for an unshared instruction stream, which could be destructively modified without causing havoc.

Other approaches to "linear-like" logic include *connection graphs* [Bawden86], *chemical abstract machines* [Berry90], *linear abstract machines* [Lafont88] and *interaction nets* [Lafont90].

We have not yet integrated arrays into our Linear Lisp, so we cannot perform imperative array updates. However, [Baker91SB] shows an efficient O(1) implementation of array updates for "single-threaded" programs. Another approach would be to incorporate *I-Structures* [Arvind89] into Linear Lisp.

Acknowledgements

We appreciate the helpful discussions with Peter Deutsch, Richard Fateman, Robert Keller, Nori Suzuki and David Wise about these concepts.

References

Abadi, M., and Plotkin, G.D. "A Logical View of Composition and Refinement". *Proc. ACM POPL 18* (Jan. 1991),323-332.

Arvind, and Nikhil, R.S. "Executing a program on the MIT tagged-token dataflow architecture". *PARLE'87, v. II*, Springer LNCS 259, 1987, 1-29.

Arvind, and Nikhil, R.S., and Pingali K.K. "I-Structures: Data Structures for Parallel Computing". *ACM TOPLAS 11*,4 (Oct. 1989),598-632.

Baker, H.G., and Hewitt, C. "The Incremental Garbage Collection of Processes". *Proc. ACM Symp. on AI & Progr. Langs., Sigplan Not. 12*,8 (Aug. 1977),55-59.

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

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

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

Baker, H.G. "Shallow Binding Makes Functional Arrays Fast". *ACM Sigplan Not. 26*,8 (Aug. 1991),145-147.

Baker, H.G. "The Boyer Benchmark at Warp Speed". ACM *Lisp Pointers V*,3 (Jul-Sep 1992), 13-14.

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

Barth, Paul S., *et al*. "M-Structures: Extending a Parallel, Non-strict, Functional Language with State". *Proc. Funct. Progr. Langs. & Computer Arch.*, LNCS 523, Springer-Verlag, Aug. 1991,538-568.

Bawden, Alan. "Connection Graphs". *Proc. ACM Conf. on Lisp & Funct. Progr.*, Camb., MA, Aug. 1986,258-265.

Beeler, M., Gosper, R.W, and Schroeppel, R. "HAKMEM". AI Memo 239, MIT AI Lab., Feb. 1972. Important items: 102, 103, 104, 149, 150, 161, 166, 172.

Berry, G., and Boudol, G. "The Chemical Abstract Machine". *ACM POPL 17*, San Francisco, CA, Jan. 1990,81-94.

Bloss, A. "Update Analysis and the Efficient Implementation of Functional Aggregates". *4'th Conf. on Funct. Progr. & Comp. Arch.*, London, Sept. 1989, 26-38.

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

Collins, G.E. "A method for overlapping and erasure of lists". *CACM 3*,12 (Dec. 1960),655-657.

Ershov, A.P. "On Programming of Arithmetic Operations". *Doklady, AN USSR 118*,3 (1958),427-430, transl. Friedman, M.D., *CACM 1*,8 (Aug. 1958),3-6.

Fateman, Richard J. "Endpaper: FRPOLY: A Benchmark Revisited". *Lisp & Symbolic Comput. 4* (1991),155-164.

Gabriel, R.P. *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

Gabriel, R.P. "The Why of Y". *ACM Lisp Pointers 2*,2 (Oct.-Dec. 1988),15-25.

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

Goto, Eiichi. "Monocopy and Associative Algorithms in an Extended Lisp". Tech. Rep. 74-03, Info. Sci. Lab., U. Tokyo, April 1974.

Goto, Eiichi, and Kanada, Yasumasa. "Hashing Lemmas on Time Complexities with Applications to Formula Manipulation". *Proc. ACM SYMSAC'76*, Yorktown Hgts., NY, 1976.

Goto, E., *et al*. "Parallel Hashing Algorithms". *Info. Proc. Let. 6*,1 (Feb. 1977),8-13.

Goto, E., *et al.* "Design of a Lisp Machine — FLATS". *Proc. ACM Lisp & Funct. Progr. Conf.*, 1982,208-215.

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

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

Herlihy, Maurice. "Wait-Free Synchronization". *ACM TOPLAS 11*,1 (Jan. 1991),124-149.

Inoue, K., *et al*. "Analysis of functional programs to detect run-time garbage cells". *ACM TOPLAS 10*,4 (Oct. 1988), 555-578.

Johnsson, T. "Efficient compilation of lazy evaluation". *Proc. 1984 ACM Conf. on Compiler Constr.*, Montreal, 1984.

Johnsson, T. "Lambda lifting: transforming programs to recursive equations". *Proc. FPCA*, Nancy, France, Springer LNCS 201, 1985,190-203.

Johnsson, T. "Parallel Evaluation of Functional Programs: The <v,G>-machine approach". *Proc. PARLE'91, v. I.*, Springer LNCS 505, Berlin, 1991, 1-5.

Jones, S.B., and Le Metayer, D. "Compile-time garbage collection by sharing analysis". *ACM Funct. Progr. Langs. & Comp. Arch.*, 1989, 54-74.

Kieburtz, Richard B. "Programming without pointer variables". *Proc. Conf. on Data: Abstraction, Definition and Structure, Sigplan Not. 11* (special issue 1976),95-107.

Kieburtz, Richard B. "The G-machine: a fast, graph-reduction evaluator". *Proc. IFIP FPCA*, Nancy, France, 1985.

Kieburtz, Richard B. "A RISC Architecture for Symbolic Computation". *Proc. ASPLOS II, Sigplan Not. 22*,10 (Oct. 1987),146-155.

Knight, Tom. "An Architecture for Mostly Functional Languages". *Proc. ACM Conf. on Lisp & Funct. Progr.*, MIT, Aug. 1986,105-112.

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

Lafont, Yves. "Interaction Nets". *ACM POPL 17*, San Franciso, CA, Jan. 1990,95-108.

Lafont, Yves. "The Paradigm of Interaction (Short Version)". Unpubl. manuscript, July 12, 1991, 18p.

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

MacLennan, B.J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,2 (Dec. 1982),70-79.

Mairson, H.G. "Deciding ML Typability is Complete for Deterministic Exponential Time". *17'th ACM POPL*, Jan. 1990, 382-401.

Mason, Ian A. *The Semantics of Destructive Lisp*. Ctr. for the Study of Lang. & Info., Stanford, CA, 1986.

Moon, D. "Garbage Collection in a Large Lisp System". *ACM Symp. on Lisp and Functional Prog.*, Austin, TX, 1984, 235-246.

Morris, J.M. "A proof of the Schorr-Waite algorithm". In Broy, M., and Schmidt, G., *Eds*. *Theoretical Foundations of Programming Methodology*. NATA Advanced Study Inst., D. Reidel, 1982,43-51.

Myers, Eugene W. "Efficient Applicative Data Types". *ACM POPL 11*, Salt Lake City, UT, Jan. 1984,66-75.

Peyton-Jones, S.L. *The Implementation of Functional Programming Languages*. Prentice-Hall, New York, 1987.

Rees, J. and Clinger, W., *et al*. "Revised Report on the Algorithmic Language Scheme". *Sigplan Notices 21*,12 (Dec. 1986),37-79.

Ruggieri, Cristina; and Murtagh, Thomas P. "Lifetime analysis of dynamically allocated objects". *ACM POPL '88*,285-293.

Schorr, H., and Waite, W.M. "An efficient machine-independent procedure for garbage collection in various list structures". *CACM 10*,8 (Aug. 1967),501-506.

Strom, R.E., *et al*. "A recoverable object store". IBM Watson Research Ctr., 1988.

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

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

Turner, D. "A New Implementation Technique for Applicative Languages". *SW—Pract.&Exper. 9* (1979),31-49.

Wadler, Philip. "Linear types can change the world!". *IFIP TC2 Conf. on Progr. Concepts & Meths.*, April 1990.

Wadler, Philip. "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.

Weizenbaum, J. "Symmetric List Processor". *CACM 6*,9 (Dec. 1963),524-544.
White, Jon L. "Address/Memory Management for a Gigantic LISP Environment, or GC Considered Harmful". *Proc. 1980 Lisp Conf.*, Stanford U., Aug. 1980,119-127.
Wilson, Paul R. *Two comprehensive virtual copy mechanisms*. Master's Thesis, U. Ill. @ Chicago, 1988.
Wilson, P.R., and Moher, T.G. "Demonic memory for process histories". *Proc. Sigplan PLDI*, June 1989.
Wilson, Paul R. "Some Issues and Strategies in Heap Management and Memory Hierarchies". *ACM Sigplan Not. 26*,3 (March 1991),45-52.
Wise, D.S., and Friedman, D.P. "The one-bit reference count". *BIT 17*,3 (Sept. 1977),351-359.
Wise, David S. "Design for a Multiprocessing Heap with On-board Reference Counting". *Proc. Funct. Progr. Langs & Computer Arch.*, Nancy, France, LNCS 201, Springer, Sept., 1985,289-304.

Appendix. A Metacircular Linear Lisp Interpreter

```
(defmacro free (x) `(setq ,x nil)) ; just for testing...

(defmacro mif (be te ee)
  (let ((vs (car (freevars be))))
    `(if-function
       #'(lambda () ,be)
       #'(lambda ,vs ,te)
       #'(lambda ,vs ,ee))))

(defun if-function (be te ee)
  (dlet* (((pval . stuff) (funcall be)))
    (if pval (apply te stuff) (apply ee stuff))))

(defmacro mcond (&rest clauses)
  (dlet* ((((be . rest) . clauses) clauses))
    (if (eq be 't) `(progn ,@rest)
      `(mif ,be (progn ,@rest)
          (mcond ,@clauses)))))

(defun matom (x) `(,(atom x) ,x))

(defun meq (x y) `(,(eq x y) ,x))

(defun meq2 (x y) `(,(eq x y) ,x ,y))

(defun msymbolp (x) `(,(symbolp x) ,x))

(defun mnull (x) `(,(null x) ,x))

(defun mcopy (x) ; Pedagogical non-primitive copy function.
  (mif (matom x) `(,x ,@x) ; primitive is allowed to copy symbol refs.
    (dlet* (((carx . cdrx) x)
            ((ncar1 . ncar2) (mcopy carx))
            ((ncdr1 . ncdr2) (mcopy cdrx)))
      `((,ncar1 ,@ncdr1) . (,ncar2 ,@ncdr2)))))

(defun massoc (x e) ; return binding, else nil.
  (mif (mnull e) (progn (free x) `(nil ,@e))
    (dlet* ((((var . val) . reste) e))
      (mif (meq2 x var) (progn (free x) `((,var ,@val) ,@reste))
        (dlet* (((nbinding . nreste) (massoc x reste)))
          `(,nbinding . ((,var ,@val) ,@nreste)))))))

(defun mevprogn (xl e)
  (dlet* (((x . xl) xl))
    (mif (mnull xl) (progn (assert (null xl)) (meval x e))
      (dlet* (((xval . ne) (meval x e))
              ((xlval . nne) (mevprogn xl ne)))
        (free xval)
        `(,xlval ,@nne)))))
```



```
(defun meval (x e)
  (mcond
   ((msymbolp x) (dlet* ((((var . val) . ne) (massoc x e)))
                   (free var)
                   `(,val ,@ne)))
   ((matom x) `(,x ,@e))
   (t (dlet* ((fn . args) x))
        (mcond
         ((meq fn 'progn) (free fn) (mevprogn args e))
         ((meq fn 'function) (free fn)
          (dlet* (((lambda) args)
                  ((nlambda . lambda) (mcopy lambda))
                  ((fvars . bvars) (freevars `(function ,nlambda) nil nil))
                  ((e1 . e2) (split fvars e)))
            `((funarg ,lambda ,e1) ,@e2)))
         ((meq fn 'funcall) (free fn)
          (dlet* ((((nfn . nargs) . ne) (mevlis args e)))
            `(,(mapply nfn nargs) ,@ne)))
         (t (dlet* (((nargs . ne) (mevlis args e)))
              `(,(mapply (symbol-function fn) nargs) ,@ne))))))))

(defun mapply (fn args)
  (mif (matom fn) (apply fn args)
    (dlet* (((ffn . rfn) fn))
      (mcond
       ((meq ffn 'lambda) (free ffn)
        (dlet* (((bvlist . body) rfn)
                ((v . ne) (mevprogn body (mpairlis bvlist args nil))))
          (assert (null ne))
          v))
       ((meq ffn 'funarg) (free ffn)
        (dlet* ((((lambda bvlist . body) ce) rfn)
                ((v . ne) (mevprogn body (mpairlis bvlist args ce))))
          (free lambda) (assert (null ne))
          v))
       (t (error "mapply: bad fn ~S" fn))))))

(defun mpairlis (vars vals e)
  (mif (mnull vars) (progn (assert (null vals)) e)
    (dlet* (((var . vars) vars)
            ((val . vals) vals))
      `((,var ,@val) ,@(mpairlis vars vals e)))))

(defun mevlis (args e)
  (mif (mnull args) (progn (assert (null args)) `(nil ,@e))
    (dlet* (((x . args) args)
            ((xval . e) (meval x e))
            ((argvals . e) (mevlis args e)))
      `((,xval ,@argvals) ,@e))))

(defun split (vars e) ; split env. into 2 segments.
  (mif (mnull vars) (progn (assert (null vars)) `(nil ,@e))
    (dlet* (((var . nvars) vars)
            ((binding . ne) (massoc var e))
            ((e1 . e2) (split nvars ne)))
      `((,binding ,@e1) ,@e2))))
```

```
(defun mmember (x ls) ; return truth value & rest of list.
  (mif (mnull ls) (progn (free x) `(nil ,@ls))
    (dlet* (((carls . ls) ls))
      (mif (meq2 x carls) (progn (free x) `(,carls ,@ls))
        (dlet* (((tval . rest) (mmember x ls)))
          `(,tval . (,carls ,@rest)))))))

(defun freevars (x bvars fvars) ; return new fvars and new bvars.
  (mcond
   ((msymbolp x)
    (dlet* (((x1 . x2) (mcopy x))
            ((x2 . x3) (mcopy x2))
            ((p1val . nbvars) (mmember x1 bvars))
            ((p2val . nfvars) (mmember x2 fvars)))
      (mif p1val (progn (free x3) `(,nfvars ,@nbvars))
        (mif p2val (progn (free x3) `(,nfvars ,@nbvars))
          `((,x ,@nfvars) ,@nbvars)))))
   ((matom x) (free x) `(,fvars ,@bvars))
   (t (dlet* (((fn . args) x))
        (mcond
         ((meq fn 'function) (free fn)
          (dlet* ((((lambda bvlist . body)) args))
            (free lambda)
            (freelistvars body `(,@bvlist ,@bvars) fvars)))
         (t (freelistvars `(,fn ,@args) bvars fvars)))))))

(defun freelistvars (xl bvars fvars)
  (mif (mnull xl) (progn (assert (null xl)) `(,fvars ,@bvars))
    (dlet* (((x . xl) xl)
            ((nfvars . nbvars) (freelistvars xl bvars fvars)))
      (freevars x nbvars nfvars))))
```