ACM *Sigsam Bulletin* **27**,4 (December 1993), 10-14.

Sparse Polynomials and Linear Logic

Henry G. Baker

Nimble Computer Corp., 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 Gabriel *FRPOLY* benchmark was rewritten in a "linear" fragment of Common Lisp and is competitive with the standard FRPOLY benchmark code. This linear FRPOLY is considerably more perspicuous than the standard code, while its running time is only 6% longer than that of the standard FRPOLY code. Linear FRPOLY recovers *all* of its garbage, and its "high water mark" space requirement is very probably smaller than that of the standard code. In the expansion of $(x+y+z+1)^{15}$, the standard FRPOLY does 48,892 new conses, while the linear FRPOLY does only 4821 new conses—i.e., it does only about 10% of the consing of the standard FRPOLY code.

We also tested versions of FRPOLY in which squarings were *not* used for exponentiation. This non-linear FRPOLY does 38,780 conses, and takes only 59% of the time of the non-linear squaring FRPOLY. The linear non-squaring FRPOLY takes only 62% of the time of the linear squaring FRPOLY, and cuts the new consing to 3988 cells. A slightly slower version cuts the new consing to 2590 cells—only 567 cells (28%) more than are used in the result.

INTRODUCTION

Two early major applications of Lisp are the *Reduce* and *Macsyma* symbolic algebra systems. Due to the complexity of algebraic expressions, the Lisp list constructed from "cons" cells provides an elegant and efficient representation for the manipulations that must be performed. Therefore, when Richard Gabriel selected a set of programs for benchmarking Lisp systems [Gabriel85], he included the *FRPOLY* program, which is a fragment of the Macsyma system that adds and multiplies polynomials in several variables. Polynomial manipulations are the heart of any symbolic algebra system, and as polynomial *multiplication* is both common and expensive, the FRPOLY benchmark raises the simple polynomial r=x+y+z+1 to the powers 2, 5, 10 and 15 by successive squaring. As the size of the answer explodes exponentially with larger powers, this expansion of $(x+y+z+1)^{15}$ provides an interesting benchmark.

"Impure" Lisp is frowned on in beginning Lisp classes, both because the student should know how much can be done without "side-effects"—e.g., assignment and list modification—and because such code can be difficult to understand [Mason86]. However, professional Lispers sometimes mangle lists in the pursuit of time or space efficiency. We were drawn to the FRPOLY benchmark because it utilizes such list "mangling". We wanted to understand its operation, to see whether its speed and space efficiency were achievable in a "linear" fragment of Lisp.

Before discussing "linear" Lisp, we first note that there are two errors in the FRPOLY code [Gabriel85,3.20.1] that cause the polynomial multiplication routine `ptimes` to return incorrect results. The first error occurs in the `pzerop` macro, which evaluates its argument twice (a linearity checker would have complained about the multiple use of this macro's argument!). The `pzerop` macro is called almost always with a trivial argument, but in the routine `ptimes3`, two lines down from the label `b`, it is called with an argument expression that has a side-effect—namely an assignment. There are several ways to fix this problem, the simplest being the following:

```
(defmacro pzerop (x) `(let ((x ,x)) (if (numberp x) (zerop x) nil)))
```

The second error occurs in the routine `ptimes3` near the label `b`. The following two lines should be deleted:

```
b  (if (or (null (cdr u)) (< (cadr u) e))              ; These 2 lines should
       (rplacd u (cons e (cons c (cdr u)))) (go e))    ; be replaced.
```

and should be replaced with:

```
b  (cond ((or (null (cdr u)) (< (cadr u) e))           ; The if should be a
          (rplacd u (cons e (cons c (cdr u)))) (go e))); cond.
```

In addition to producing incorrect results, these errors cause problems in comparing benchmark results because they change the amount of work that is performed. For example, the expansion of $(x+y+z+1)^{15}$ should perform 48,892 conses,[1] but the buggy code performs only 33,574 conses.

A few remarks about the programming style used in the standard FRPOLY code are in order. The style of programming used for most of the functions is traditional recursive Lisp programming, as found in any beginning book on Lisp. The polynomial *addition* routines do not utilize any side-effects, nor do they take advantage of traditional Lisp's fluid/dynamically-scoped variables.

The style of programming used within the polynomial *multiplication* routines, however, is dramatically different—it relies heavily on both dynamically-scoped variables and side-effects to variables and data structures. According to Gabriel: "[this benchmark] is programmed in an unusual and somewhat unpleasant programming style"

[1]The number of conses (79,140) listed in the "Meter for (Bench 15)" table of [Gabriel85,p.249] may be incorrect. These additional conses may be used in bignum arithmetic in the expansion of $(100000*(x+y+z+1))^{15}$.

[Gabriel85,3.20.2], an analysis with which almost every professional Lisp programmer would agree. The `ptimes3` routine is a nightmare for either a person or a compiler to understand.

The `ptimes3` code was written by Bill Martin in about 1968 when the available list space was severely limited by the PDP-10's 256Kword address space, and before optimizing Lisp compilers became available [Fateman91]. Bill's goals were the fast execution consistent with minimum space [Fateman91]. Minimizing the maximum number of cons cells in use at any instant (the "high water mark") seemed more of a concern than minimizing the number of calls to the `cons` routine. Unfortunately, measuring the exact "high water mark" of a non-linear Lisp computation is problematical, since it theoretically requires performing a garbage collection for every single call to the `cons` routine [Baker92BB]. Thus, we cannot tell for sure whether Bill's `ptimes3` actually achieved his goals.

A SHORT TUTORIAL ON "LINEAR" LISP

"Linear" Lisp is a style of Lisp programming 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*`, `multiple-value-bind`, etc. A linear language requires more work from the programmer to make explicit any copying or deletion, but 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 does not have a dangling reference problem.*

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 [Baker92LLL] 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 [Baker92LLL] for a definition of `dup`). The `square` function follows:

```
(defun square (x)
  (multiple-value-bind (x x-prime) (dup x)   ; Dylan bind syntax [Shalit92] would be prettier.
    (* x x-prime)))
```

Conditional expressions such as `if`-expressions require a bit of sophistication. Since only one of the "arms" of the conditional will be executed, we relax the "one-occurrence" linearity condition to allow a reference in both arms.[2] 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]).

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.[3]

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 append (x y)
  (if-null x (progn (kill x) y)                                ; trivial kill.
    (dlet* (((carx . cdrx) x))
      (cons carx (append cdrx y)))))                 ; cons will be optimized.
```

---

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

[3]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.


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

We "warm up" for FRPOLY by programming a simple univariate "dense" polynomial multiplication routine written in a linear fragment of Lisp, in which the polynomials are simply lists of integer coefficients.

```
(defun pplus (x y)                                              ; Return x+y.
  (if-null x (progn (kill x) y)                               ; trivial kill.
    (if-null y (progn (kill y) x)                             ; trivial kill.
      (dlet* (((x0 . x) x) ((y0 . y) y)) ; decompose 1 cell from both x and y.
        (cons (+ x0 y0) (pplus x y))))))          ; cons will be optimized.
(defun pctimes (x0 y)                                          ; Return x0*y.
  (if-null y (progn (kill x0) y)                              ; trivial kill.
    (multiple-value-bind (x0 x0-prime) (dup x0)                ; trivial dup.
      (dlet* (((y0 . y) y))                        ; decompose 1 cell from y.
        (cons (* x0 y0) (pctimes x0-prime y))))))    ; cons will be optimized.
(defun ptimes (x y)                                             ; Return x*y.
  (if-null x  (progn (kill y) x)                ; possibly nontrivial kill.
    (dlet* (((x0 . x) x))                          ; decompose 1 cell from x.
      (if-null x (progn (kill x) (pctimes x0 y))              ; trivial kill.
        (multiple-value-bind (y y-prime) (dup y)            ; nontrivial dup.
          (pplus (pctimes x0 y)
                 (cons 0 (ptimes x y-prime)))))))))   ; cons will be optimized.
(defun psquare (x)
  (multiple-value-bind (x x-prime) (dup x)
    (ptimes x x-prime)))
(defun pexptsq (x n)
  (if-zerop n (progn (kill x) (kill n) (cons 1 nil))        ; nontrivial kill.
    (if-evenp n (psquare (pexptsq x (floor n 2)))
      (multiple-value-bind (x x-prime) (dup x)
        (ptimes x (psquare (pexptsq x-prime (floor n 2))))))))
```

Our Linear Lisp FRPOLY is completely analogous to this code, except that it manipulates a "sparse" representation which allows coefficients to recursively be polynomials in other variables.

RESULTS FROM LINEAR FRPOLY BENCHMARK AND DISCUSSION

The polynomial addition routines of FRPOLY required only minor modifications to be put into linear form. The polynomial multiplication routines had to be completely rewritten, however. The new style of these routines is analogous to that shown for univariate polynomial multiplication, above.

We also had to program the basic `dup`, `kill`, `dlet*` and `cons` routines for our linear fragment of Common Lisp. Since the basic intuition behind Linear Lisp is that *all arguments to a function are consumed in the production of results from the function*. This means that `dlet*` not only *binds* names to portions of a substructure, but also *recycles cons cells that it has destructured*. We provided a private freelist for these recycled cons cells, and a modified `cons` routine utilizes these recycled cells first, prior to calling the system `cons` routine. The `kill` routine dissolves a structure down to its atoms, and recycles all of its constituent cons cells. The `dup` routine copies its argument using the modified `cons` routine, and returns both the original and the copy as results. By utilizing these four primitives and the linearity restriction, one is guaranteed that no cons cells can be lost.[4] This is because linearity is essentially the same as reference counting [Collins60], which additionally enforces a no-cycles policy. Of course, our reference counts are always exactly 1, so that no actual reference count updating is ever needed![5]

For our FRPOLY benchmark, we measured `(pexptsq r n)`, for `n` from `2` to `15`. Consider now `n=15`. We counted the number of conses in the input r=x+y+z+1 (15 cons cells), the number of conses in the output r^{15} (2038 cons cells), the number of calls to the *system* `cons` routine, and the number of recycled cells left on the private

[4]One cannot utilize non-local exits such as `return-from`, `throw`, `go`, etc., however, because these violate the "occur-once" linearity constraint for many of the bindings passed on the stack. One could conceivably use `unwind-protect` to ensure the linearity constraints, but it could be quite messy. Perhaps a new kind of "protect" form is needed for Linear Lisp. (Schemers beware: a linear Scheme can have only return-once continuations.)

[5]A "behind-the-scenes" reference count scheme can be used to implement fast `dup`'s and `kill`'s [Baker92LLL], but we did not utilize such a scheme in our implementation of FRPOLY.

freelist after FRPOLY has finished. The numbers tallied: #output - #input + #free - #consed = 0, indicating that all cells were accounted for.

In our first linear version of FRPOLY with `n=15`, the system `cons` routine was called 8375 times and 6352 recycled cells were left on the private freelist at the end. Although 8375 was only a factor of 4 larger than the 2038 cells in the output structure, we felt that we could do better. We subsequently located several places in the code where `dup` was called just before a recursive call which performed a test and terminated the recursion. In the process of termination, it called `kill` on the same structure which it had just `dup`'ed. By moving the recursion termination test up into the previous recursion, the extra `dup` and `kill` were eliminated, which significantly reduced the calls to the system `cons` routine. After these changes, FRPOLY called `kill` on a non-atom only once—for the original 15 input cells. The other 112,617 reused cells were recycled by `dlet*`. FRPOLY called the system `cons` routine only 4821 times, and left 2798 recycled cells on the private freelist. There were 3831 calls to `dup`, with a total of 47,692 cells copied. The `dup` per-call statistics were: mean 12.45 cells, standard deviation 35.1 cells and max 1717 cells.

By simply interchanging the arguments to the last call to `ptimes` in the `pexptsq` routine, the following statistics were observed. The total number of new conses was still 4821, `kill` was called once for the 15 input cells, and `dlet*` recycled 114,468 cells. There were now 4448 calls to `dup`, with a total of 48,926 cells copied. The `dup` per-call statistics were: mean 11.0 cells, standard deviation 20.06 cells, and max 346 cells. The interchanged time was 1.8% slower.

In order to put these cell numbers in perspective, we must examine the workings of polynomial manipulation a bit closer. In the course of computing r^{15}, `pexptsq` computes $r^{14}=r^7 \bullet r^7$. Although space for one of the copies of r^7 is reused to compute r^{14}, the space for the other copy is not recycled until the end. Therefore, `pexptsq` must require space for at least r, r^7 and r^{14}, all at the same time. Although these space requirements still do not account for the total space used, they indicate that we are getting near the lower bound.

Our first timings were significantly worse than the standard FRPOLY benchmark. However, we quickly isolated a major cause of this inefficiency—the manipulations of the global variable which held the private freelist. Through the use of machine-specific "subprimitives" of our Lisp system, we were able to reduce the cost of accessing a global variable to a single machine instruction. Since this freelist variable was extensively accessed, we achieved a major improvement through this change. Obviously, a true Linear Lisp implementation would utilize a fast register to point to the top of the freelist, rather than a global memory cell.

Another major improvement in performance was achieved through the replacement of the `rplaca` and `rplacd` instructions by additional machine-specific "subprimitives". This change allowed the `cons` from our private freelist to be slightly faster than the system `cons`.

We exploited a valuable optimization suggested by [Wakeling91]—to have the destructuring pattern-match "compiler" defer the generation of code to put cells back onto the global free-list until the end of the body of this form. This laziness allows any consing seen within the body to utilize these garbage cells immediately rather than allocating them from the freelist at run-time. This optimization saved about 20% of the running time. Unfortunately, this optimization works only within a single function, so larger functions can be somewhat better optimized. For this reason, we inlined `pcoefadd` and `psimp`.

Minor improvements in performance were obtained through various optimizations in the `dup` routine, which may do extensive work, even though it is seldom called. Additional improvements came from making sure that `dup` and `kill` were called only for non-atoms, and from other similar restructurings of the code.

The net result of these improvements is that our latest linear version of FRPOLY runs only 6% slower than the standard benchmark on our machine (an 4 Mbyte Apple Macintosh Plus with a 16MHz Radius 68020 processor card) running Coral Common Lisp version 1.2. We believe that this timing is very competitive with the standard benchmark, since our linear FRPOLY timings include garbage collection, while the standard FRPOLY timings do not include garbage collection.

We also ran tests with an exponentiation routine that did *not* use squaring—i.e., it used straight-forward successive multiplications [Fateman74] [Fateman91]. We considered a version in which the smaller argument to `ptimes` was first ("normal") and the smaller argument to `ptimes` was second ("reversed"). Both the "normal" and "reversed" versions of the non-linear FRPOLY did 38,780 new conses, or about 79% of the conses of the squaring non-linear FRPOLY. The running time on the non-linear FRPOLY was 59% of the squaring non-linear FRPOLY for both "normal" and "reversed" versions ([Fateman91] got 50% on a similar test).

Using multiplications in our linear FRPOLY instead of squarings reduced the space requirements. The total number of new conses dropped to 3988 for "normal" and 2590 for "reversed". In the "normal" run, there were *no* calls to `kill` non-atoms, and 72,611 cells were recycled by `dlet*`, while in the "reversed" run, there were also no calls to `kill`, and 79,709 cells were recycled by `dlet*`. In the "normal" run, there were 1358 calls to `dup`, with a total of 25,823 cells copied. The per-call statistics for `dup` in the "normal" run were: mean 19.02 cells, standard deviation

82.92 cells and max 1717 cells. In the "reversed" run, there were 3724 calls to `dup`, with a total of 30,555 cells copied. The per-cell statistics for `dup` in the "reversed" run were: mean 8.2 cells, standard deviation 16.2 cells, and max 283 cells. Although the "normal" required more space (3988 new conses) than the "reversed" (2590 new conses), it was 10% faster. This "normal" linear run took only 65% of the time of the squaring non-linear FRPOLY!

The most surprising thing to us about all of these timings was not that the linear timings were slower than the standard FRPOLY timings, but that the linear timings were even remotely competitive. After all, the general presumption is that polynomial manipulations require a lot of sharing, and that extensive copying should be significantly slower. Furthermore, the linear FRPOLY cannot just simply *examine* a data structure—it must actually *destroy* it and then *reconstruct* it. That this examination by destruction and reconstruction is at all competitive was quite a surprise to us. On newer machines with a true data cache, such examination by destruction and reconstruction should be even more competitive than on our cacheless machine.

CONCLUSIONS

Contrary to some previous reports [Wakeling91], a linear programming style is not necessarily expensive, and for a popular benchmark, the Common Lisp FRPOLY benchmark for symbolic algebraic manipulation, our linear FRPOLY was competitive in space and time with the standard non-linear FRPOLY. Furthermore, the linear code is considerably easier to understand, because it does not use fluid/dynamic variables or (visible) side-effects.

Unlike the suggestions of some [Wakeling91], we feel that integers themselves *should* be linear—especially in a computer algebra system. This is because the manipulation of large-precision integers (i.e., *bignums*) constitutes a significant fraction of the work of these systems, and it is essential that any garbage created be quickly recovered. In fact, we believe that a bignum package would be one of the easiest and most efficacious uses of linearity in a symbolic algebra system.

While an efficient linear coding style occasionally copies a large structure all at once, it rarely lets a large structure go all at once. The vast majority of the recycled cells are reclaimed one at a time by the primitive which performs a destructuring binding of new names to portions of a structure. (This behavior may shed some light on the question of the efficiency of "generational" garbage collection, as well as on the best mechanisms to update reference counts in a reference-counting system.) This copy-all-at-once but deallocate-one-at-a-time behavior sheds light on the maximum space requirement for a symbolic computation which does not utilize sharing, since the elimination of all explicit copying can reduce this maximum space requirement. A non-copying coding style is different from a more usual style, in that many functions reconstruct and return their some of their arguments as additional results so that explict copies will not be required.

We feel that the particular list data structure used in FRPOLY may not be optimum for a computer algebra system. A sparse polynomial representation should probably use larger nodes which can hold an exponent, a coefficient, a reference count, and a pointer to the next node. While true reference counting on Lisp's small 2-element nodes is probably too expensive, it might be efficient for 4-element nodes. Destructuring these larger nodes would make good use of a computer's "load multiple" capability, an optimization Lisp implementations have so far not utilized.

A somewhat surprising result was our conclusion that an exponentiation algorithm using successive squarings is *more* expensive in both time and space than an exponentiation algorithm which simply does repeated multiplications. Exponentiation by squaring may eventually be cheaper, but the break-even exponent is larger than 15.

REFERENCES

Abramsky, S. "Computational interpretations of linear logic". *Theor. Comp. Sci. 111* (1993), 3-57.
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.
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.
Fateman, R. "On the Computation of Powers of Sparse Polynomials". *Studies Appl. Math. 53*,2 (1974), 145-155.
Fateman, R. "FRPOLY: A Benchmark Revisited". *Lisp & Symbolic Comput. 4* (1991), 155-164.
Gabriel, R.P. *Performance and Evaluation of Lisp Systems*. MIT Press, Camb., MA 1985.
Girard, J.-Y. "Linear Logic". *Theoretical Computer Sci. 50* (1987),1-102.
Mason, I.A. *The Semantics of Destructive Lisp*. CSLI LN 5, Stanford, CA, 1986.
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.
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.

