DAGWOOD
A System for Manipulating Polynomials Given by Straight-Line Programs*

Timothy S. Freeman† *Gregory M. Imirzian*‡ *Erich Kaltofen*‡ *Lakshman Yagati*§

† Dept. Computer Science, C.M.U., Pittsburgh, PA 15213
‡, § Dept. Computer Science, R.P.I., Troy, NY 12181
‡ Mathematical Sciences Research Inst., 1000 Centennial Dr., Berkeley, CA 94720

Abstract. We discuss the design, implementation, and benchmarking of a system that can manipulate symbolic expressions represented by their straight-line computations. Our system is capable of performing rational arithmetic on, evaluating, differentiating, taking greatest common divisors of, and factoring polynomials in straight-line format. The straight-line results can also be converted to standard sparse format. We show by example that our system can handle problems for which conventional methods lead to excessive intermediate expression swell.

* This material is based upon work supported by the National Science Foundation under Grant No. DCR-85-04391 and by an IBM Faculty Development Award. Computational resources during our stay at Berkeley were supported in part by the Army Research Office under Grant No. DAAG29-85-K-0070 through the Center for Pure and Applied Mathematics at the University of California. This paper appears in the ACM Transactions on Mathematical Software, Vol. 14, No. 3, September 1988, Pages 218-240. A preliminary version of this paper also appears in the Proceedings of the 1986 ACM Symposium on Symbolic and Algebraic Computation, 169-175.

1. Introduction

Object representation, and in particular the representation of multivariate rational functions, is one of the most important issues in the design of computer algebra systems. In 1971 Moses [19] classified the different approaches of representation. He observed that canonical representation sometimes leads to exponential intermediate expression swell, a classical example being that of expanding symbolic determinants. One way to combat this problem is to allow the sharing of common subexpressions, and almost all major systems have some facilities to accommodate this. With the advent of probabilistic zero-testing [21], [9], and [8], and hash coding [18] and [4], canonical representations for solving the zero identity problem were not required any longer. Moreover, should the expressions after being manipulated lead to a sparse polynomial answer, that answer could be retrieved by Zippel's sparse polynomial interpolation procedure [27] or, where applicable, by a sparse Hensel lifting technique [28] and [12]. However, relatively few problems could be solved with expressions in non-canonical representation, compared to the development of important algorithms for computing greatest common divisors (GCDs) and factorizations of canonically represented multivariate polynomials. We refer to Stoutemyer [22] for a comparison of several canonical representation schemes for sparse multivariate polynomials.

One of the most compact representations for a multivariate polynomial or rational function is that by a *straight-line program*. In short, a polynomial is represented by a directed acyclic multiple computation graph for it, e.g. a Gaussian elimination sequence encoded in a list of single assignments would represent a polynomial determinant. Arithmetic on expressions in straight-line format is almost trivial, since all one has to do is add an additional assignment, or a node in the computation graph. Common subexpressions need to be computed only once and can be referred to over and over again. The difficulty arises when one tries to manipulate straight-line programs in some usual way, say when one tries to compute the GCD of two polynomials given by a straight-line program. Fortunately, we could show [16] that one can find in probabilistic polynomial-time another straight-line program that in fact computes the GCD of such inputs. Other fundamental straight-line results, which are probabilistic polynomial-time constructible, are those for the irreducible factors of a polynomial given by a straight-line program and for the numerators and denominators of a rational function [14].

This paper describes a Lisp-based system, called Dagwood, that realizes the straight-line representation for polynomials and rational functions and implements those operations that we have proven to be random polynomial-time computable. In particular, our system allows rational arithmetic, evaluation, differentiation, GCD computation, factorization, and conversion to sparse format of polynomials given by straight-line programs. Various useful utility functions are also provided, such as determining the degrees of a polynomial given by a straight-line program or finding another straight-line program that computes its coefficients in an individual variable. Several of the main algorithms, such as the GCD, factorization, and the conversion algorithm to sparse format are of the Monte-Carlo kind, which means the algorithms always terminate in polynomial-time but may with controllably small probability return an incorrect answer. It should be also said that the running times depend polynomially on the degrees of the inputs as unary integers. Clearly, a straight-line program can encode a polynomial whose degree is exponential in the input size. But for such polynomials the GCD problem is NP-hard for even the sparse representation [20], and hence a degree restriction has to be made.

It is not obvious at all that the GCD and factorization problems are feasible for polynomials in straight-line representation. To prove our point, consider a seemingly easier operation, that of computing partial derivatives. Letting

$$f(x_{1,1},\ldots,x_{n,n},y_1,\ldots,y_n) = \prod_{i=1}^{n}(\sum_{j=1}^{n} x_{i,j}y_j)$$

Valiant [24] observes that

$$\frac{\partial^n f}{\partial y_1 \cdot\cdot\cdot \partial y_n} = \text{permanent}(\begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & & \vdots \\ x_{n,1} & \cdots & x_{n,n} \end{bmatrix}).$$

Clearly, f can be computed by a straight-line program of length $O(n^2)$, whereas the computation of the permanent is by Valiant's results #P-hard. Therefore it is believed that no straight-line program of length $n^{O(1)}$ exists that computes the permanent, and hence the intermediate expression swell for iterated partial derivatives is inherent even for the straight-line representation. Note that for less space-efficient representations this phenomenon has been observed without appealing to the theory of NP-completeness [3].

Aside from the just mentioned negative result, several efficient straight-line program transformations have been developed in the context of computational algebraic complexity. Most notably are the method by Strassen [23] for eliminating divisions from computations for polynomials, the method by Baur and Strassen [1] for computing all first partial derivatives, and the probabilistic equivalence test of straight-line programs [10]. One of the first results for polynomials represented by straight-line programs is the efficient computation of their factor degree pattern by von zur Gathen [6].

As a final output to an involved problem the straight-line representation may be illegible to the user. Fortunately, the sparse interpolation algorithm allows to convert such represented answers to sparse polynomials. In our implementation the complexity of the sparse interpolation algorithm also depends on a given maximal number of allowed terms in the sparse answer. If the straight-line polynomial has more terms, the algorithm indicates that without running an exponential number of steps. One might argue that if one is interested in quantitative results such as the sparse representation of the output polynomial, rather than qualitative results such as the number of irreducible factors of the answer, which could be determined by the algorithm in [6], then sparse interpolation can be employed without ever constructing the intermediate straight-line result. Unfortunately, all known sparse Hensel lifting algorithms [28], [7] and [13], spoil this hope for the following reason. Assume one of the irreducible factors of a sparse lifting problem is dense, as may happen even for sparse inputs [7], §5. Then the sparse Hensel lifting procedures take time polynomially proportional to the number of terms in all factors including the dense ones. However, it is conceivable that one is only interested in those factors that are sparse and only wants to identify the dense ones. The straight-line factorization algorithm together with our version of the sparse interpolation algorithm allows just that, and the running time of the combined algorithms is also not affected by selection of unlucky evaluation points.

We view Dagwood as a prototype to assess the practicality of our algorithms for computer algebra. Nonetheless, our functions are not just a mere implementation of our algorithms, as we have also attempted to achieve high efficiency.

2. Top Level Description

We now describe how Dagwood can be invoked from within a standard computer algebra system. Currently, our system contains a set of routines to interface with Macsyma. This allows users to interact with Dagwood from the Macsyma environment, in particular to input expressions in Macsyma format. However, interfaces to other Lisp-based computer algebra systems may be provided in the future.

As noted in the introduction, most of our algorithms are probabilistic in nature. Some of our functions are deterministic or “Las Vegas,” which means that an error exit may be taken and a failure message produced with small probability. Of course, by calling such a function repeatedly an answer will eventually be returned. However, the functions StraightDegree, StraightGCD, StraighttoSparse, StraightFactor, StraightOpt3, and StraightNumDen in the list below are “Monte Carlo,” which means that with small probability an incorrect result is returned. That probability is algorithmically controllable, and a global flag **epsilon** could be implemented to limit the maximal probability with which any of these functions returns an incorrect result. Currently the probability of producing an incorrect answer is approximiately 10^{-7}, and to date such an event has not occurred.

In the following descriptions of our Macsyma-callable functions, *stprog* in an argument list means that the argument should be a straight-line program and *polyexp* means the argument should be a polynomial expression. Acceptable polynomial expressions are:

(i) A Macsyma-format polynomial.

(ii) A straight-line program for a polynomial.

(iii) “ ’determinant(M)”, where M is a Macsyma-format matrix whose entries are polynomial expressions. The single quote prevents evaluation of the determinant.

(iv) Any combination of (i), (ii), and (iii) using addition, subtraction, multiplication, division, and integer exponentiation.

By (iv) straight-line programs can be added, say, by the Macsyma infix ‘+’.

PolytoStraight (*polyexp*)
Return a straight-line program whose last instruction evaluates to *polyexp*. What follows now is a Macsyma example of this function call. For readers unfamiliar with Macsyma we mention that lines labeled ‘(c.)’ are user input lines and lines labeled ‘(d.)’ are the corresponding results. If input lines are terminated by a ‘$’ the output lines are suppressed. The Macsyma assignment operator is the colon ‘:’. The timings were obtained on a Symbolics 3670 Lisp Machine.

```
(c4) p1:(x^2+2*x*y+y^2)/(x+y)$
Time= 16.7 msecs.
(c5) p2:(x-y)*p1;
Time= 0.0 msecs.
                                2            2
                   (x - y) (y  + 2 x y + x )
(d5)               -------------------------
                                y + x
(c6) sp1:polytostraight(p1)$
Time= 16.7 msecs.
(c7) sp2:polytostraight((x-y)*sp1);
Time= 33.3 msecs.
v1 := 0
v2 := x
v3 := -1
v4 := y
v5 := v3 * v4
 ...(18 instructions total.)
```

StraightPrint (*stprog*)
Print a readable form of *stprog*. The global variable **stprintlimit** (default: 5) indicates how many straight-line assignments will be explicitly printed. The following continues the previous Macsyma example.

```
(c8) stprintlimit:20$
Time= 16.7 msecs.
(c9) straightprint(sp2)$
v1 := 0
v2 := x
v3 := -1
v4 := y
v5 := v3 * v4
v6 := v2 + v5
v7 := 1
v8 := v2 + v4
v9 := v7 / v8
v10 := v2 * v2
v11 := 2
v12 := v2 * v4
v13 := v11 * v12
v14 := v4 * v4
v15 := v13 + v14
v16 := v10 + v15
v17 := v9 * v16
v18 := v6 * v17
Time= 283.0 msecs.
```

StraightDegree (*stprog*)
Probabilistically determine the total degree of the polynomial given by *stprog*. The algorithm works briefly as follows. If $f(x_1,\ldots,x_n)$ is the polynomial given by *stprog*, the routine returns the degree in x_1 of $g(x_1) = f(x_1, b_2x_1,\ldots, b_nx_1)$ where the b_i are random integers. This degree is determined by comparing at a random integer a the value of $g(a)$ to the values at a of increasingly higher degree interpolation polynomials for g until agreement is found. Therefore, the function may also return a value if *stprog* does not compute a polynomial, or it my 'hang.' In order to test whether *stprog* computes a polynomial, use StraightNumDen discussed below instead. In certain circumstances, the total degree has already been probabilistically determined, such as by a previous call to StraightDegree, in which case that value is retrieved from *stprog*.

```
(c10) straightdegree(sp2);
Time= 133.0 msecs.
(d10)                    2
```

StraightProfile (*stprog*)
Display the total degree, input variables, counts by instruction type, and total length of *stprog*. A list of these values is returned.

```
(c11) straightprofile(sp2)$
Program Profile:
    Instruction counts:
        input          2
        constant       4
        plus           4
        minus          0
        times          7
        quotient       1
        Total Length  18
    Total degree:  2
    Input variables:  x y
Time= 433.0 msecs.
```

Evalmodp (*stprog, p*)
Evaluate *stprog* modulo the integer p and return the value of the last instruction of *stprog*. Each input variable in *stprog* must be bound to a rational number.

RatEval (*stprog, bound*)
After a suitable modulus is selected, *stprog* is evaluated via Evalmodp and then the rational value of the last instruction of *stprog* is recovered from its modular image and returned. *Bound* is a bound on the absolute value of the numerator and denominator of the result. If *bound* is too low, the routine usually reports an error but will sometimes return a bad result. Cf [16],. Evaluation algorithm.

```
(c12) x:3/4$
Time= 0.0 msecs.
```

```
(c13) y:1/4$
Time= 0.0 msecs.
(c14) rateval(sp2,10);
Time= 50.0 msecs.
                                   1
(d14)                              -
                                   2
```

StraightDiff (*stprog*, *var* [, *order*])
Construct a straight-line program for the *order*-th derivative with respect to *var* of the rational function given by *stprog*, where *var* is a variable symbol. If *order* is omitted, it is defaulted to 1. A new Macsyma example follows illustrating this and the two following functions. For a higher order derivative, see the example with StraightNumDen below.

```
(c6) p : ratsimp((x+y^2+z^3)^2*(2*x^2+3*y^3+5*z^5));
Time= 267.0 msec.
        11          2           8         3       2    6
(d6) 5 z   + (10 y  + 10 x) z  + (3 y  + 2 x ) z
      4          2       2   5         5          3       2  2        3    3
 + (5 y  + 10 x y  + 5 x ) z  + (6 y  + 6 x y  + 4 x  y  + 4 x ) z
       7         5       2  4       2  3      3  2       4
 + 3 y  + 6 x y  + 2 x  y  + 3 x  y  + 4 x  y  + 2 x
(c7) dp : straightdiff(polytostraight(p),x);
Time= 450.0 msec.
v1 := 0
v2 := 2
v3 := x
 ...(127 instructions total.)
```

StraightGCD (*polyexp*$_1$,..., *polyexp*$_n$ [, db=*degree-bound*])
Return a straight-line program for the GCD of the input polynomials using the algorithm described in [16]. *Degree-bound* is a bound on the total degrees of the input polynomials. If omitted a bound is computed using StraightDegree. In the following examples, the flag **verbose** is set so that additional information is printed during the evaluation of the procedures, which appears here in italics. However, we have taken the liberty to remove some of the produced trace information in certain places.

```
(c8) verbose: true$
Time= 0.0 msec.
(c9) /* GCD of p and dp should be the squared factor of p */
sqfac : straightgcd(p,dp)$
Composite program length: 165
Computed Total Degree is 11
Done calculating coefficients.
    program length = 997
Degree of first polynomial = 11
Degree of second polynomial = 11
Degree of remainder is 7
Degree of remainder is 6
Degree of remainder is 5
Degree of remainder is 4
Degree of remainder is 3
Degree of remainder is −1
Done computing GCD.
Time= 3800.0 msec.
(c10) straightprofile(sqfac)$
Program Profile:
   Instruction counts:
      input         3
      constant      11
      plus          397
      minus         84
      times         682
      quotient      7
      Total Length  1184
   Total degree:  3
   Input variables:  x y z
Time= 433.0 msec.
```

StraighttoSparse (*stprog*, *height-bound* [, db=*degree-bound*] [, terms=*term-bound*] [, norm=no])
Compute the sparse representation of the polynomial given by *stprog* using the sparse interpolation algorithm described in [27] and [15], and return a Macsyma-format polynomial as the result. *Height-bound* is a bound for the absolute values of the numerators and denominators of the rational coefficients of the result, *degree-bound* is a bound for the degree of *stprog*, and *term-bound* is the maximum number of terms allowed in the answer. The procedure returns *false* if the input polynomial has probably more than *term-bound* many monomials. If *degree-bound* is omitted, a bound is probabilistically determined. If *term-bound* is omitted, no interrupt occurs even if the polynomial is dense. If norm=no is omitted, the resulting sparse answer will be multiplied by a rational such that the coefficient of some term is 1. This is useful for converting GCDs or factors to sparse representation, since the straight-line programs for those can compute a large scalar multiple of the normalized result. One should realize that this normalization cannot take place before the conversion to sparse.

```
(c11) straighttosparse(sqfac,10);
Beginning sparse interpolation.
Program length: 1184
Normalization ON
Computed Total Degree is 3
Interpolating variable $x
The degree bound for this variable is 1.
```

```
We now have 1 unknown nonzero coefficients and 0 known coefficients.
The system has 2 equations.
Interpolating variable $y
The degree bound for this variable is 2.
We now have 2 unknown nonzero coefficients and 0 known coefficients.
The system has 6 equations.
Interpolating variable $z
The degree bound for this variable is 3.
We now have 3 unknown nonzero coefficients and 0 known coefficients.
The system has 9 equations.
Done with sparse interpolation.
Time= 9020.0 msecs.
                         3     2
(d11)                   z  + y  + x
```

StraightCoeff (*stprog, var*)
Returns a list of straight-line programs for the coefficients in *var* of the polynomial given by *stprog*, where *var* is a variable symbol. The list has length the degree in *var* plus 1, whose *i*-th entry is the coefficient of x^{i-1} of *stprog.* The coefficients are computed in one of two ways [16]:

(1) Direct computation: Loop through *stprog* and calculate the representation of each instruction as a power series in *var*.

(2) Interpolation: Construct a straight-line program representation for the coefficients of the Lagrangian interpolation polynomial for the polynomial given by *stprog*. Each function evaluation in the interpolation formula is represented by a copy of *stprog* in which *var* is replaced by a constant.

It turns out that method (1) is asymptotically inferior to method (2). However, in practice method (1) often leads to a shorter resulting program than method (2). Therefore, StraightCoeff estimates the length of the result for both methods and calls the one that gives a smaller estimated lenght. We observed that the length estimates made are quite accurate.

```
(c4) /*5 by 5 general Vandermonde determinant*/
p: 'determinant(matrix([1,x1,x1^2,x1^3,x1^4],
                              [1,x2,x2^2,x2^3,x2^4],
                              [1,x3,x3^2,x3^3,x3^4],
                              [1,x4,x4^2,x4^3,x4^4],
                              [1,x5,x5^2,x5^3,x5^4]));
Time= 16.7 msecs.
```

```
                         [            2    3    4 ]
                         [ 1  x1  x1   x1   x1   ]
                         [                        ]
                         [            2    3    4 ]
                         [ 1  x2  x2   x2   x2   ]
                         [                        ]
(d4)    determinant([            2    3    4 ])
                         [ 1  x3  x3   x3   x3   ]
                         [                        ]
                         [            2    3    4 ]
                         [ 1  x4  x4   x4   x4   ]
                         [                        ]
                         [            2    3    4 ]
                         [ 1  x5  x5   x5   x5   ]
(c5) c: straightcoeff(polytostraight(p),x1)$
Direct estimate:        920
Interpolation estimate: 678
Constructing interpolation formula ...
   evaluation points: 0 1 2 3 4
Done computing coefficients.
Time= 1820.0 msecs.
```

StraightFactor (*polyexp, height-bound* [, db=*degree-bound*])
A list of pairs [*factor, multiplicity*] is returned, where *factor* is a straight-line program for an irreducible factor of multiplicity *multiplicity*. *Height-bound* should be a bound for the absolute value of the numerators and the common denominator of the rational coefficients of the input polynomial. The optional *degree-bound* should be a bound for the total input degree and is probabilistically determined if omitted. Cf [14]. and [15]. This routine calls the Macsyma univariate factorization procedure. In particular the improvements by Wang [26] have proved important for the efficiency of the entire process. The following example factors the coefficient of x_1^4 in the previous example. Notice that by considering a minor expansion along the first row we conclude that that coefficient is a sub-Vandermonde determinant and therefore factors as $(x_3 - x_2)$ $(x_4 - x_2)$ $(x_5 - x_2)$ $(x_4 - x_3)$ $(x_5 - x_3)$ $(x_5 - x_4)$.

```
(c6) f:straightfactor(c[5],100)$
Beginning factorization.
Exponents of univariate factors: (1 1 1 1 1 1)
Starting partial fraction computation ...
Done computing partial fractions.
Program length = 3711
Beginning Lifting.
   Parts of degree 1 lifted.  7876
Lifting Complete.
Time= 66800.0 msecs.
(c7) straighttosparse(f[3][1],10);
Program length: 11614
```

```
Time= 98400.0 msecs.
(d8)               x3 - x2
```

StraightOpt3 (*stprog*)
Optimize the argument. This function probabilistically condenses *stprog* by removing all instructions that compute previously computed functions. Briefly, the algorithm works as follows. All instructions in the program are evaluated at random values for the indeterminates modulo a random prime. Then a binary search tree is built to sort their values. If a value of a new instruction is found in an existing leaf, the corresponding instructions are assumed to compute the same function and the new one is eliminated from the program. The algorithm is Monte Carlo and its running time is that of sorting, essentially $O(l \log(l))$, where l is the length of the *stprog*, since with high probability the search tree is well balanced. The following example optimizes the result of the previous factorization.

```
(c8) f3 : straightopt3(f[3][1])$
3185(27%) instructions saved.
Time= 77400.0 msecs.
(c9) straighttosparse(f3,10);
Program length: 8429
Time= 59100.0 msecs.
(d9)               x3 - x2
```

StraightNumDen (*ratexp*, *num-degree-bound* [, *den-degree-bound*])
Returns a list of two straight-line programs for the numerator and denominator of *ratexp*, which is like a *polyexp* but can denote a rational function. *Num-degree-bound* should be a bound for the total degree of the numerator and *den-deg-bound* should be a bound for the denominator of *ratexp*. If the latter is omitted it is defaulted to *num-degree-bound*. This function can be used to test whether *ratexp* computes a polynomial by applying StraightDegree to the denominator program. For the algorithm description refer to [14] and [16], §8.

```
(c1) p:y/x+1/(x-y);
Time= 233.0 msecs.
                                        y       1
(d1)                                    - + -----
                                        x   x - y
(c2) q:polytostraight(p)$
Time= 117.0 msecs.
(c3) r:straightdiff(q,x,3)$
Time= 300.0 msecs.
(c4) r1:straightopt3(r)$
6(16%) instructions saved.
Time= 817.0 msecs.
(c5) b:straightnumden(r1,10,10)$
Beginning separation into Numerator and Denominator...
NumDenMonitor OFF
Variable translation done.
```

```
Taylor expansion upto degree 20 done.
Program length = 797
Numerator and Denominator to be normalised....
Performing reverse translation....
Degree of the numerator = 5
Degree of the denominator = 8
Time= 1980.0 msecs.
(c6) b2:straightopt3(b[2])$
149(9%) instructions saved.
Time= 24900.0 msecs.
(c7) d:straighttosparse(b2,100000000);
Beginning sparse interpolation.
Program length: 1381
Normalization ON
Computed Total Degree is 8
Interpolating variable $x
The degree bound for this variable is 8.
We now have 1 unknown nonzero coefficients and 0 known coefficients.
The system has 9 equations.
Interpolating variable $y
The degree bound for this variable is 4.
We now have 4 unknown nonzero coefficients and 1 known coefficients.
The system has 14 equations.
Done with sparse interpolation.
Time= 477000.0 msecs.
                          4  4                6  2            8
                         x  y      5  3    3 x  y      7     x
(d7)                   - ----- + x   y  - ------- + x  y - --
                           4                 2               4
(c8) factor(d7);
                                         4       4
                                        x  (y - x)
(d8)                                  - -----------
                                             4
```

StraightLinSolve (*equ-list*, *var-list*)
Returns a list of straight-line programs that solve the system of linear equations in *equ-list* with respect to the variabls in *var-list*. Both argument lists are as for the Macsyma LINSOLVE function, except that StraightLinSolve currently only supports square non-singular systems. For linear systems built from straight-line programs themselves, see the function StraighterLinSolve. The following example exhibits several of the previous functions as well.

```
(c4) eq1: x1+x2+x3+x4=1;
Time= 0.0 msecs.
(d4)                     x4 + x3 + x2 + x1 = 1

(c5) eq2: a*x1+b*x2+c*x3+d*x4=y;
```

```
Time= 16.7 msecs.
(d5)              d x4 + c x3 + b x2 + a x1 = y

(c6) eq3: a^2*x1+b^2*x2+c^2*x3+d^2*x4=y^2;
Time= 100.0 msecs.
                2       2       2       2       2
(d6)           d  x4 + c  x3 + b  x2 + a  x1 = y
(c7) eq4: a^3*x1+b^3*x2+c^3*x3+d^3*x4=y^3;
Time= 33.3 msecs.
                3       3       3       3       3
(d7)           d  x4 + c  x3 + b  x2 + a  x1 = y
(c8) s:straightlinsolve([eq1,eq2,eq3,eq4],[x1,x2,x3,x4])$
Time= 1150.0 msecs.
(c9) s1:straightopt3(s[1])$
18(19%) instructions saved.
Time= 1020.0 msecs.
(c10) ss:straightnumden(s1,3,3)$
Beginning separation into Numerator and Denominator...
Taylor expansion upto degree 6 done.
Program length = 970
Degree of the numerator = 3
Degree of the denominator = 3
Time= 1580.0 msecs.
(c11) ss1:straightopt3(ss[1])$
172(15%) instructions saved.
Time= 13400.0 msecs.
(c12) ss1f: straightfactor(ss1,10^30)$
Beginning factorization.
Program length: 937
Univariate factors:
(8 a - 49) (27 a - 32) (68 a + 53)
Exponents of univariate factors: (1 1 1)
Bound: 496812601600000000000000000000000000
Constant factor: ((rat simp) 5 320399444309405638574187471)
Program length = 3779
Beginning Lifting.
   Parts of degree 1 lifted.   10596
Lifting Complete.
Time= 174000.0 msecs.
(c13) for i: 1 thru length(ss1f) do
          print(straighttosparse(ss1f[i][1],10000))$
Beginning sparse interpolation.
Program length: 14401
y - d
y - b
y - c
Time= 635000.0 msecs.
```

```
(c14) ss2:straightopt3(ss[2])$
172(15%) instructions saved.
Time= 17700.0 msecs.
(c15) ss2f: straightfactor(ss2,10^30)$
Time= 165000.0 msecs.
(c16) for i: 1 thru length(ss2f) do
          print(straighttosparse(ss2f[i][1],10000))$
Beginning sparse interpolation.
Program length: 14473
a − b
a − d
c − a
```

StraighterLinSolve (*matrix*, *vector*)
Returns a list of straight-line programs that solve $matrix^{-1} \times vector$, where *matrix* is a square non-singular matrix whose entries are straight-line programs, and *vector* is a one column matrix with the same number of rows and straight-line entries as well.

```
(c5) a1:1$
(c6) a2:−1*a$
(c7) a3:a^2$
(c8) a4:b$
(c9) a5:b^2$
(c10) a6:c$
(c11) a7:c^2$
(c12) a8:y$
(c13) a9:y^2$
(c14) b1:polytostraight(a1)$
(c15) b2:polytostraight(a2)$
(c16) b3:polytostraight(a3)$
(c17) b4:polytostraight(a4)$
(c18) b5:polytostraight(a5)$
(c19) b6:polytostraight(a6)$
(c20) b7:polytostraight(a7)$
(c21) b8:polytostraight(a8)$
(c22) b9:polytostraight(a9)$
(c23) m:matrix([b1,b1,b1],[b2,b4,b6],[b3,b5,b7])$
(c24) n:matrix([b1],[b8],[b9])$

(c25) s:straighterlinsolve(m,n)$
Time= 450.0 msecs.

(c26) s1:straightopt3(s[1])$
9(22%) instructions saved.
Time= 550.0 msecs.
(c27) ss:straightnumden(s1,3,3)$
Beginning separation into numerator and denominator...
```

```
Taylor expansion upto degree 6 done.
Program length = 316
Degree of the numerator = 2
Degree of the denominator = 2
Time= 883.0 msecs.

(c28) ss1:straightopt3(ss[1])$
65(16%) instructions saved.
Time= 9780.0 msecs.
(c29) ss1f: straightfactor(ss1,10^30)$
Time= 87700.0 msecs.
(c30) for i: 1 thru length(ss1f) do
          print(straighttosparse(ss1f[i][1],10000))$
y - b
c - y
Time= 64900.0 msecs.

(c31) ss2:straightopt3(ss[2])$
65(16%) instructions saved.
Time= 7120.0 msecs.
(c32) ss2f: straightfactor(ss2,10^30)$
Time= 54300.0 msecs.
(c33) for i: 1 thru length(ss2f) do
          print(straighttosparse(ss2f[i][1],10000))$
c + a
b + a
Time= 95100.0 msecs.
```

3. More Examples

Example 1: In this factorization example one of the factors is a dense polynomial. The straight-line method finds the sparse factor relatively quickly, while the standard factorization method is bogged down by computing the dense factor as well.

```
(c1) p : (a+b+c+d+w+x+y+z)^4-a*b*c*d;
Time= 33.3 msecs.
                                                   4
(d1)          (z + y + x + w + d + c + b + a)  - a b c d
(c2) q : a^2+z^2;
Time= 0.0 msecs.
                                  2    2
(d2)                             z  + a
(c3) spq : polytostraight(p*q)$
Time= 83.3 msecs.
(c4) sf : straightfactor(spq,100)$
Time= 11700.0 msecs.
(c5) straightprofile(sf[1][1])$
Program Profile:
   Instruction counts:
      input          8
      constant       56
      plus           476
      minus          53
      times          662
      quotient       0
      Total Length  1255
   Total degree:  2
   Input variables:  a z b c d w x y
Time= 817.0 msecs.
(c6) sf1 : straightopt3(sf[1][1])$
374(29%) instructions saved.
Time= 4930.0 msecs.
(c7) straighttosparse(sf1,10,terms=3);
Time= 10300.0 msecs.
                                  2    2
(d7)                             z  + a
(c8) straighttosparse(sf[2][1],10,terms=3);
Term bound exceeded.
Time= 3780.0 msecs.
(d8)                              false
(c9) factor(p*q)$
Time= 641000.0 msecs.
```

Example 2: For this example we took the GCD of the 8 by 8 Vandermonde determinant $\det([x_i^j]_{1 \le i \le 8, 0 \le j \le 7})$ and $x_1^{27} - x_2^{27}$. The straight-line program for the GCD took 2 minutes to compute and was 21,174 instructions long. It took 5 minutes to convert it to the sparse polynomial $x_1 - x_2$. Macsyma used a sparse modular GCD scheme only after expanding the determinant and it took 10 hours to arrive at the solution.

Example 3: Not in all cases can the straight-line representation compete with the sparse methods. The following factorization example is Claybrook's polynomial 17 [5]. Macsyma uses Wang's improved factorizer [25] and is an order of magnitude faster. Notice that all factors are sparse and there are only four variables, conditions that generally favour the sparse method.

```
(c1) p : (z-r-x^2*y+w-x*y^2)*(-1-x+y-z^2*w-z+z*w^2+w);
Time= 467.0 msecs.
                2     2                      2     2
(d1) (z - x y  - x  y + w - r) (- w z  + w  z - z + y - x + w - 1)
(c2) sf : straightfactor(polytostraight(p),10)$
Time= 12600.0 msecs.
(c3) straighttosparse(sf[1][1],10);
Time= 15300.0 msecs.
                           2     2
(d3)              - w z  + w  z - z + y - x + w - 1
(c4) factor(p);
Time= 383.0 msecs.
                  2     2                      2     2
(d4) - (z - x y  - x  y + w - r) (w z  - w  z + z - y + x - w + 1)
```

Example 4: This example shows a factorization with multiplicities. It also displays the fact that normalization in the conversion to sparse format is not so crucial as for GCD results. This is a small example and the Macsyma factorizer necessarily wins.

```
(c1) p : (2*u^2+3*v+5*w+7*x+11*y+13*z+17)^2;
Time= 33.3 msecs.
                                                   2      2
(d1)          (13 z + 11 y + 7 x + 5 w + 3 v + 2 u  + 17)
(c2) q : (z^2-y-x-w-v-u+11)^3;
Time= 33.3 msecs.
                         2                             3
(d2)                  (z  - y - x - w - v - u + 11)
(c3) sf : straightfactor(polytostraight(p*q),1000)$
Time= 14300.0 msecs.
(c4) straighttosparse(sf[1][1],100);
Time= 22900.0 msecs.
                                                   2
               13 z   11 y   7 x       3 v   2 u    17
(d4)           ---- + ---- + --- + w + --- + ---- + --
                5      5      5         5     5     5
(c5) sf[1][2];
Time= 0.0 msecs.
(d5)                               2
```

```
(c6) factor(p*q);
Time= 450.0 msecs.
                                              2       2
(d6) (13 z + 11 y + 7 x + 5 w + 3 v + 2 u  + 17)
                                               2                            3
                                            (z  - y - x - w - v - u + 11)
(c7) straighttosparse(sf[1][1],100,norm = no);
Time= 20300.0 msecs.
                                                    2
(d7)         13 z + 11 y + 7 x + 5 w + 3 v + 2 u  + 17
```

Example 5: In this GCD example of several polynomials the straight-line approach and Macsyma's sparse modular approach perform approximately equal.

```
(c1) p : u^2*v^2*w^2+x^2*y^2*z^2;
Time= 283.0 msecs.
                              2   2   2     2   2   2
(d1)                         x   y   z  + u   v   w
(c2) q1 : (u+2*v+3*w+4*x+5*y+6*z)^6;
Time= 66.7 msecs.
                                                          6
(d2)               (6 z + 5 y + 4 x + 3 w + 2 v + u)
(c3) q2 : (u^3*v^3+w^3*x^3+y^3*z^3);
Time= 33.3 msecs.
                             3   3      3   3     3   3
(d3)                        y   z  + w   x  + u   v
(c4) q3 : 'determinant(matrix([u+v,w+x,y+z],
                              [(u+z)^2,(v+y)^2,(w+x)^2],
                              [(z+y)^3,(x+w)^3,(v+u)^3]));
Time= 333.0 msecs.
                            [   v + u       x + w       z + y    ]
                            [                                    ]
                            [          2           2           2 ]
(d4)          determinant([  (z + u)     (y + v)     (x + w)    ])
                            [                                    ]
                            [          3           3           3 ]
                            [  (z + y)     (x + w)     (v + u)    ]
(c5) sgcd : straightgcd(p*q1,p*q2,p*q3)$
Time= 4150.0 msecs.
(d5) straighttosparse(sgcd,10);
Time= 41000.0 msecs.
                              2   2   2     2   2   2
(d6)                         x   y   z  + u   v   w
(c7) gcd(p*q1,p*q2,p*determinant(part(q3,1)));
Time= 47600.0 msecs.
```

```
                               2   2   2      2   2   2
(d7)                          x   y   z   + u   v   w
```

Example 6: The next example again makes a strong case for the straight-line representation. The determinant to be factored is extremely dense, but obviously contains a sparse factor, which is easily found by our algorithms. We note that the Macsyma took 1.7 hours to factor the expanded determinant.

```
(c1) p : 'determinant(matrix([w+x+y+z,a+b+c,u+v,0],
                            [(a-x-y-z)^2,(u-b-c)^2,(d-w)^2,0],
                            [(a+b+c+d)^3,(x+y+z)^3,(u+v)^3,0],
                            [(u+z)^5,(x+d)^5,(a+w)^5,x^2+y^2+z^2]));
Time= 250.0 msecs.
               [  z + y + x + w      c + b + a     v + u          0       ]
               [                                                          ]
               [                 2               2          2             ]
               [(- z - y - x + a)   (u - c - b)   (d - w)          0       ]
(d1)determinant([                                                          ])
               [                3               3          3              ]
               [ (d + c + b + a)    (z + y + x)   (v + u)          0       ]
               [                                                          ]
               [             5                5          5   2     2     2]
               [      (z + u)           (x + d)    (w + a)   z   + y   + x ]
(c2) sf : straightfactor(polytostraight(p),1000)$
Time= 37100.0 msecs.
(c3) straightlength(sf[1][1]);
Time= 100.0 msecs.
(d3)                                  11565
(c4) sfo : straightopt3(sf[1][1])$
1811(15%) instructions saved.
Time= 110000.0 msecs.
(c5) straighttosparse(sfo,10,terms=3);
Time= 111000.0 msecs.
                                     2     2     2
(d5)                                z   + y   + x
(c6) straighttosparse(sf[2][1],10,terms=3);
Term bound exceeded.
Time= 28900.0 msecs.
(d6)                                   false
(c7) factor(d1)$
Time= 6120000.0 msec.
```

Example 7: As in the previous example, this one considers the factorization of a symbolic determinant. Here it is not obvious at all what the factors would be. This computation is one of several carried out jointly with K. Johnson for his investigations on determinants of Moufang loops [11].

```
                  [ 1    1    1    1    1    1    1    1  ]
                  [                                       ]
                  [ x2   x1   x4   x3   x6   x5   x8   x7 ]
                  [                                       ]
                  [ x3   x4   x1   x2   x8   x7   x5   x6 ]
                  [                                       ]
                  [ x4   x3   x2   x1   x7   x8   x6   x5 ]
(d7)              [                                       ]
                  [ x5   x6   x7   x8   x1   x2   x3   x4 ]
                  [                                       ]
                  [ x6   x5   x8   x7   x2   x1   x4   x3 ]
                  [                                       ]
                  [ x7   x8   x5   x6   x4   x3   x1   x2 ]
                  [                                       ]
                  [ x8   x7   x6   x5   x3   x4   x2   x1 ]
(c8) polytostraight('determinant(d7))$
Time= 1550.0 msecs.
(c9) straightopt3(d8)$
62(14%) instructions saved.
Time= 3880.0 msecs.
(c10) straightfactor(d9,1000)$
Univariate factors:
(75 x2 - 457) (169 x2 - 51) (229 x2 - 37)
            2                                2
   (94405 x2  + 11958 x2 - 20049) (114645 x2  - 69158 x2 + 7881)
Exponents of univariate factors: (1 1 1 1 1)
Program length = 3474
Beginning lifting.
   Parts of degree 1 lifted.   8936
   Parts of degree 2 lifted.   7418
Lifting complete.
Time= 105000.0 msecs.
(c11) straightopt3(d11[1][1])$
1000(5%) instructions saved.
Time= 172000.0 msecs.
(c12) straighttosparse(d12,1000);
Time= 278000.0 msecs.
(d12)          x8 + x7 + x6 + x5 - x4 - x3 - x2 - x1
(c13) straighttosparse(d11[4][1],1000);
Time= 386000.0 msecs.
          2             2     2             2     2             2
        x8            x7    x6            x5    x4            x3
(d13) - --- + x7 x8 - --- + --- - x5 x6 + --- + --- - x3 x4 + ---
         2             2     2             2     2             2
```

```
                                                              2                 2
                                                            x2                x1
                                                        - --- + x1 x2 - ---
                                                             2                 2
(c14) length(expand(determinant(d3)));
Time= 1800000.0 msecs.
(d14)                          2792
```

4. Data Structure for Straight-Line Programs

Straight-line programs are represented internally by triples of the form

(*label instruction-list last-instruction*).

Label indicates that an object is a straight-line program, and its format depends on the top-level system being used. Under Macsysma, *label* is always '(straight simp)'. *Instruction-list* is a list of straight-line instructions. Instruction formats are described below. *Last-instruction* is a pointer to that instruction in *instruction-list* whose value the program represents. Instructions following this last instruction are irrelevant for the computation of that value. This field is included so that one instruction list can be shared by several different straight-line programs.

There are three types of straight-line instructions:

(*value-cell* **input** *input-variable*),
(*value-cell* **constant** *scalar*),
and (*value-cell opcode* var_1 var_2).

Opcode is one of {**plus**, **minus**, **times**, **quotient**}, *input-variable* is a variable symbol, *scalar* is either an integer or a form for a rational number (e.g., '((rat simp) *m n*)' in Macsyma), and var_1 and var_2 are pointers to previous instructions. The *value-cell* is used by the system to store information about an instruction, for example the value of that instruction during an evaluation or the total degree of the polynomial given by the instruction.

5. Assessment

It is evident that the creation of a system for manipulating polynomials in straight-line representation was possible only after fundamental operations such as GCD and factoring were known to be feasible. However, we realize that additional effort and ingenuity is required to make a polynomial-time process practically efficient. It should be clear that our system is geared towards objects that cannot be manipulated by sparse techniques due to their denseness. This implies that our system should be applied to large problems. We observe also that the complexity of all our algorithms, except the sparse interpolation algorithm, is independent of the number of indeterminates in the input polynomials. Therefore, Dagwood is very competitive on problems with many variables. Dagwood in its current version only allows rational coefficients. There are no theoretical problems in extending the system to allow coefficients from algebraic extensions of the rationals represented by polynomials in the generators of these extensions modulo their minimal polynomial, and to coefficicients from finite fields. Also, recently new methods for the StraighttoSparse procedure were discovered [2], [29], and [17], and should significantly speed that process.

We must warn, however, that the straight-line representation is not yet a panacea for intermediate expression swell. Currently, our main procedures such as the GCD or factorization routines return a straight-line answer whose length is at least the length of the input program times the input degree. This growth, although polynomial, makes it virtually impossible to stack our routines back to back in the process of an algebraic computation. There are, of course, possible attacks against this problem. One is to retain additional information in the straight-line answer. The factorization procedure, for instance, needs to compute the coefficients of an individual variable, and this is exactly the way a straight-line GCD is determined. Therefore, if we were to factor a straight-line GCD, the step of retrieving the coefficients of a variable could be skipped, provided that additional information were also returned by the GCD routine. Another attack is to try to optimize the straight-line programs after they get constructed. One somewhat trivial optimization scheme is to convert to sparse and back to straight-line representation. In light of our examples leading to sparse answers this idea should not be underestimated. Another optimization scheme is used in the function StraightOpt3 with worthwhile results. Optimization, however, definitely warrants future research.

In Moses's terminology [19] the straight-line representation belongs to the “new left” for it is non-canonical and permits naturally the sharing of subexpressions. However, our algorithms operate entirely within this representation and Dagwood therefore also exhibits the traits of a “radical” one. It is our belief that our approach to a certain extent unifies the advantages of both, the new left covering a larger class of inputs and the radical supplying well-understood algorithms.

Acknowledgement: We like to thank Richard Fateman for his insightful help with Macsyma.

References

1. Baur, W. and Strassen, V., “The complexity of partial derivatives,” *Theoretical Comp. Sci.*, 22, pp. 317-330 (1983).

2. Ben-Or, M. and Tiwari, P., “A deterministic algorithm for sparse multivariate polynomial interpolation,” Manuscript (October 1987).

3. Caviness, B. F. and Epstein, H. I., “A note on the complexity of algebraic differentiation,” *Inf. Proc. Lett.*, 7, pp. 122-124 (1978).

4. Char, B. W., Geddes, K. O., Gentleman, W. M., and Gonnet, G. H., “The design of MAPLE: A compact, portable, and powerful computer algebra system,” *Proc. EUROCAL ’83, Springer Lec. Notes Comp. Sci.*, 162, pp. 102-115 (1983).

5. Claybrook, B. G., “A new approach to the symbolic factorization of multivariate polynomials,” *Artificial Intelligence*, 7, pp. 203-241 (1976).

6. Gathen, J. von zur, “Irreducibility of multivariate polynomials,” *J. Comp. System Sci.*, 31, pp. 225-264 (1985).

7. Gathen, J. von zur and Kaltofen, E., “Factoring sparse multivariate polynomials,” *J. Comp. System Sci.*, 31, pp. 265-287 (1985).

8. Gonnet, G. H., “Determining equivalence of expressions in random polynomial time,” *Proc. 16th ACM Symp. Theory Comp.*, pp. 334-341 (1984).

9. Heintz, J. and Schnorr, C. P., “Testing Polynomials which are easy to compute,” *Monographie de L’Enseignement Mathématique*, 30, pp. 237-254, Imprimerie Kundig, Gen`eve (1982).

10. Ibarra, O. H. and Moran, S., “Probabilistic algorithms for deciding equivalence of straight-line programs,” *J. ACM*, 30, pp. 217-228 (1983).

11. Imirzian, G. M., Johnson, K., and Kaltofen, E., “Factoring determinants of Latin squares,” in preparation (1987).

12. Kaltofen, E., “Sparse Hensel lifting,” *Proc. EUROCAL ’85, Vol. 2, Springer Lec. Notes Comp. Sci.*, 204, pp. 4-17 (1985).

13. Kaltofen, E., “Computing with polynomials given by straight-line programs II; Sparse factorization,” *Proc. 26th IEEE Symp. Foundations Comp. Sci.*, pp. 451-458 (1985).

14. Kaltofen, E., “Uniform closure properties of p-computable functions,” *Proc. 18th ACM Symp. Theory Comp.*, pp. 330-337 (1986).

15. Kaltofen, E., “Factorization of polynomials given by straight-line programs,” *Math. Sci. Research Inst. Preprint*, 02018-86, Berkeley, CA (1986). To appear in: ‘‘Randomness in Computation,’’ Advances in Computing Research, S. Micali ed., JAI Press Inc., Greenwich, CT, January 1987.

16. Kaltofen, E., “Greatest common divisors of polynomials given by straight-line programs,” *J. ACM*, 35, 1, pp. 231-264 (1988).

17. Kaltofen, E. and Yagati, Lakshman, “An improved sparse multivariate polynomial interpolation algorithm,” Manuscript, Dept. Computer Sci., Rensselaer Polytechnic Institute (February 1988).

18. Martin, W. A., “Determining the equivalence of algebraic expressions by hash coding,” *J. ACM*, 18, pp. 549-558 (1971).

19. Moses, J., “Algebraic simplification: A guide for the perplexed,” *Commun. ACM,* 14, pp. 548-560 (1971).

20. Plaisted, D. A., “Sparse complex polynomials and polynomial reducibility,” *J. Comp. System Sci.,* 14, pp. 210-221 (1977).

21. Schwartz, J. T., “Fast probabilistic algorithms for verification of polynomial identities,” *J. ACM,* 27, pp. 701-717 (1980).

22. Stoutemyer, D. R., “Which polynomial representation is best?,” *Proc. Third MACSYMA Users’ Conference,* pp. 221-243, General Electric, Schenectady, New York (1984).

23. Strassen, V., “Vermeidung von Divisionen,” *J. reine u. angew. Math.,* 264, pp. 182-202 (1973). (In German).

24. Valiant, L., “Reducibility by algebraic projections,” *L’Enseignement mathématique,* 28, pp. 253-268 (1982).

25. Wang, P. S., “An improved multivariate polynomial factorization algorithm,” *Math. Comp.,* 32, pp. 1215-1231 (1978).

26. Wang, P. S., “Early detection of true factors in univariate polynomial factorization,” *Proc. EUROCAL ’83, Springer Lec. Notes Comp. Sci.,* 162, pp. 225-235 (1983).

27. Zippel, R. E., “Probabilistic algorithms for sparse polynomials,” *Proc. EUROSAM ’79, Springer Lec. Notes Comp. Sci.,* 72, pp. 216-226 (1979).

28. Zippel, R. E., “Newton’s iteration and the sparse Hensel algorithm,” *Proc. ’81 ACM Symp. Symbolic Algebraic Comp.,* pp. 68-72 (1981).

29. Zippel, R. E., “Interpolating polynomials from their values,” Manuscript, Symbolics Inc. (January 1988).