Linear Algebra using Maple's `LargeExpressions` Package

J. Carette[1], Wenqin Zhou[2], D.J. Jeffrey[2], M.B. Monagan[3]

[1] McMaster University
Hamilton, ON, Canada, L8S 4L8
carette@mcmaster.ca

[2] University of Western Ontario
London ON, Canada, N6A 5B7
{wzhou7, djeffrey}@uwo.ca

[3] Simon Fraser University
Burnaby, B.C. Canada, V5A 1S6
monagan@cecm.sfu.ca

Abstract

The package `LargeExpressions` has been available in Maple for a number of years, but it is not well known. It provides tools for managing large expressions. In this paper, we describe a new application of this tool to the LU factoring of matrices. We describe a function that factors a matrix and expresses the results using a hierarchical representation. As part of the LU factoring, we introduce several strategies for pivoting, veiling an expression and zero-recognition in our function. All these strategies can be chosen based on the application. The new function is very flexible and much faster than the existing `LUDecomposition` command in Maple. Results of benchmark calculations are given.

1 Introduction

One of the attractions of Maple is that it allows users to tackle large problems. However, when users undertake large-scale calculations, they often find that expression swell can limit the size of the problems they can solve [15]. Typically, users might meet two types of expression swell: one type we can call *inherent* expression swell, and the other *intermediate* expression swell. Intermediate expression swell describes the case in which a calculation temporarily creates large expressions, *en route* to a small final result. An example that will be familiar to many readers is the calculation of a factor common to two polynomials (in other words a GCD calculation). In many cases, the factor being sought is quite small, but during its calculation, much larger expressions are generated. When people notice intermediate expression swell in a mathematical calculation, they often wonder whether they took 'the long way around' and whether a shorter calculation is possible. Sometimes this is so, and a short cut exists, but in many cases it is just the way the calculation has to go. In this paper, however, we are more concerned with inherent expression swell. By this we mean the solution of problems that result in large expressions, even when the calculations are complete. For particular problems, finding bounds on expression size can provide estimates of the amount of memory needed, or of the CPU time required to perform a given calculation. Bounds on problem size can be determined empirically (for example by a statistical survey) or theoretically (by worst case and best case analyzes). If it seems possible to perform a calculation, there remains the problem of programming it successfully in Maple. This last problem is the one addressed here.

For many problems which lead to large expressions, it is useful to control the expressions by hiding their values under user-defined labels. This allows compact representation of the results as a computation sequence, generated from the natural hierarchy of the problem [6]. The ideas of hierarchical representation of data are not confined to computer algebra. In recent years, engineering designs have become so complex that the design process can only be made feasible with the aid of hierarchical design concepts. As will be shown below, the expansion of a hierarchical expression or design generally results in a substantial increase in the size of the data. Thus, investigations have been made on how to take advantage of a hierarchical structure while processing it[2]. A recent example is the numerical integration of a DAE system while keeping it in a hierarchical representation [3].

Related ideas have appeared in the literature and in software systems already, under a variety of names, such as computation sequences and straight-line programs. Some of these related works, such as Freeman [10] and Kaltofen [11], concentrate on the manipulation of computation sequences or straightline programs to compute (e.g.) greatest common divisors of polynomials defined by such sequences. Zippel [13] shows how to use sparse interpolation to convert a computation sequence into a more standard representation of a polynomial. Other work, such as is embodied in Maple's 'optimize' command [14] or the special purpose programs of Budgell and El Maraghy [12], shows how to turn very large expressions, once generated, into more compact and useful computation sequences. Monagan and Monagan [9] designed a Code Generation Package for manipulating programs, including automatic differentiation, code optimization, and complexity analysis of a program; their application was to an optimization problem from computer vision. Their package is available in Maple as the `codegen` package. In [14], there is a discussion of computation sequences and automatic code generation with signature functions.

In 1997, Corless *et al* [7] applied large expression management to two perturbation calculations in fluid mechanics. That paper concentrated on interactively generating an appropriate computation sequence from a natural hierarchy of the problem, which is discovered as the computation proceeds. The approach has the advantage that the sequences are natural to the problem at hand, and their forms can be controlled by the user. Also because the simplifications are introduced early in a calculation, their benefits, particularly with regard to quicker processing and smaller memory requirements, can be felt throughout the rest of the calculation.

Here, the management of large expressions will be considered in the context of linear algebra. A simple example of the problems we face is obtained by typing

```
> with(LinearAlgebra):
> A := Matrix(6,6,symbol=m):
> (P, L, U) := LUDecomposition(A);
```

which gives results that are very lengthy. Using the MAPLE `length` function, we find that the lengths of the matrices L and U are 45029 and 86471. This is because Maple expands the polynomials in the numerators and denominators of the LU decomposition, and the determinant of the matrix A, which appears in the LU decomposition has 6! terms. If you want to compute an LU decomposition for larger matrices, you will have problems with both output size and computation time. Similarly, for other linear algebra operations, such as Gaussian elimination, computing the symbolic determinant of a matrix, solving a system of linear equations and so on, there are similar expression swell problems. Thus the hierarchical representation method presented in this paper can also be helpful and efficient for solving these problems.

In this paper, we discuss hierarchical representations and the MAPLE package `LargeExpressions`. One reason for the neglect of `LargeExpressions` has been the fact that it provides only the low-level tools `Veil` and `Unveil`. Here we show how to implement a higher-level function from these basic tools. In addition, we discuss the implementation of a LU symbolic decomposition algorithm with different strategies.

2 The package `LargeExpressions`

This package was added to MAPLE in 2001, and offers two main functions: `Veil` and `Unveil`.

```
Calling Sequence
    Veil[K]( complicated_expression )
    Unveil[K]( expressions_with_Ks, n )

Parameters
    K - unassigned name to use as a label
    complicated_expression - expression
    expressions_with_Ks - expression that has been veiled
    n - positive integer representing the level of unveiling, or
        infinity, meaning all levels.
```

The routine `Veil` is used to hide information, such as some complicated expression. `Unveil` reveals the previously `Veil`ed information. Both commands take an index which specifies the label to use; multiple labels can be present in an expression and manipulated independently. Here we give a simple example to illustrate how to use `Veil` and `Unveil` to flexibly express an expression in a hierarchical presentation.

Example 1: Consider an expression such as

$$Z = \log(\sin(x^2 + y^2)) + \sin(x^2 + y^2) + \cos(x^2 + y^2) \ .$$

By using the `Veil` command, we get its hierarchical representation as follows.

```
> with(LargeExpressions):
> Veil[K](x^2+y^2);
                              K[1]
> Veil[K](sin(K[1]));
                              K[2]
> Z := log(K[2]) + K[2] + cos(K[1]);
                Z := ln(K[2]) + K[2] + cos(K[1])
```

The full expression tree can always be printed out using the `Unveil` command.

```
> for i to LastUsed[K] do K[i]=Unveil[K](K[i]) od;
                         K[1] = x^2 + y^2
                         K[2] = sin(x^2+y^2)
```

Remark 1. *The `LargeExpressions` package allows users to represent their expressions in a controlled way. Users can choose the length of expression which should be veiled and define when to veil the expressions during their generations. Different strategies can be applied according to the different tasks.*

Remark 2. *Commands from `LargeExpressions` can work during expression generation to avoid intermediate expression swell problems. This is in contrast to using the `optimize` command, which only works when we have the whole expression already computed and in memory. If we cannot compute the expression, we cannot use `optimize`. So `LargeExpressions` prevents too large expressions appearing.*

Remark 3. *Veiled expressions resist unwanted simplification.*

For example, if we have an expression like

$$Z = (1 + x)^{31} - 1;$$

If the expression $1 + x$ is generated first, the `LargeExpressions` package can be used to veil it, which will resist expansion by `simplify`.

```
> Veil[K](x+1);
                              K[1]
> Z := simplify(K[1]^31-1);
                         Z := K[1]^31 - 1
```

If we apply the `simplify` command first to Z, hoping for a shorter expression, we usually cannot return to the original compact representation. Therefore it is important to veil expressions during their generation.

Remark 4. *With the* `Unveil` *command, we can verify the correctness of a compact expression at any time. We can isolate any global variable $K[i]$, and* `Unveil` *it independently. This facility can be very useful for users to evaluate any expression in which they are interested.*

3 LU Factoring with Large Expression Management

Solving systems of linear equations $Ax = b$ is central in scientific computation, where A is a coefficient matrix, b is a vector which specifies the right-hand side of the system of equations, and x is a vector of unknown values [1]. A common procedure of solving these systems is to factor the matrix A into the product of a lower-triangular matrix L and an upper-triangular matrix U such that $A = LU$; the solution is then found by forward and backward substitution. LU decomposition is used to solve dense systems of linear equations, which are found in applications [5] such as airplane wing designs, radar cross-section studies, supercomputer bench marking, etc. [4].

We modified the standard code for LU decomposition to include veiling and to use a probabilistic zero test. We also generalized the options for selecting pivots and added an option to specify a veiling strategy. One can see [19] for even more design points, and a general design strategy, for this class of algorithms. The algorithm in high-level pseudo-code is:

```
Get maximum_column, maximum_row for matrix A For current_column
from 1 to maximum_column
  for current_row from current_column to maximum_row
      Check element for zero.
      Test element for being "best" pivot
      Veil pivot [invoke Veiling strategy]
      move pivot to diagonal, recording interchanges.
      row-reduce matrix A with veiling strategy
      store multipliers in L
    end do:
```

```
  end do:
return permutation_matrix, L, reduced matrix A
```

The function has been programmed with the following calling sequence.

```
   LULEM(A, K, p, Pivoting, Veiling, Zerotesting)

Parameters
   A           - square matrix
   K           - unassigned name to use as a label
   p           - prime
   Pivoting    - decide a pivot for a column
   Veiling     - decide to veil an expression or not
   Zerotesting - decide if the expression is zero.
```

3.1 Pivoting Strategy

The current MAPLE `LUDecomposition` function selects one of two pivoting strategies on behalf of the user, based on data type. Thus, at present,
`> LUDecomposition(<<12345,1>|<1,1>>);`

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1/12345 & 1 \end{bmatrix}, \begin{bmatrix} 12345 & 1 \\ 0 & 12344/12345 \end{bmatrix}$$

even though it is more attractive to write

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 12345 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 0 & -12344 \end{bmatrix}$$

If the matrix contains floating-point entries, partial pivoting is used.
`> LUDecomposition(<<1,12345.>|<1,1>>);`

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1. & 0. \\ (8.1)10^{-5} & 1. \end{bmatrix}, \begin{bmatrix} 12345. & 1 \\ 0 & 0.99992 \end{bmatrix}$$

Since we wished to experiment with different pivoting strategies, we made it an option. Rather than make up names, such as 'partial pivoting' or 'non-zero pivoting', to describe strategies, we allow the user to supply a function which takes 2 arguments. The function returns true if the second argument is a preferred pivot to the first argument. For example, the preferred pivoting strategy for the example above (choose the smallest pivot) can be specified by the function `(p1,p2)->evalb(abs(p2)<abs(p1))`. In a symbolic and veiling context there are a number of conceivable strategies which one might wish to try. These can be based on operation count, size of expression or number of indeterminants. However, the definition of LU factors only allows pivoting on one column, so no form of full pivoting is offered.

3.2 Veiling Strategy

In the same spirit of experimentation, we have used a function to specify a veiling strategy. This function takes one argument and returns true if the expression should be veiled. The current `LargeExpressions` package, for example, follows a strategy of ignoring integers. Thus an integer, however large, cannot be veiled at present. Similarly, integer content is extracted from expressions before veiling. Rather than make these decisions in advance, we leave them to the declaration of a veiling-strategy function.

Of particular interest is the 'granularity' of the HR, namely whether one veils every pairwise operation, or whether one waits until an expression of a pre-determined size is allowed to accumulate. Another possibility is to veil sums of two or more terms because it is expansion of products of sums of terms that generally causes expression swell of formulae. In the former case, the HR would look similar to a straight-line program as defined in [18]. For our experiments, we have based our strategies on the Maple `length` command, as being a convenient measure of expression complexity.

3.3 Zero Test Strategy

We need to test for zero when finding pivots. This can also help us to simplify our expressions, if needed. During the LU factoring, we use signatures to perform this test quickly (more precisely, in random polynomial time). The basic idea of signatures [16, 14] is to evaluate a formula f modulo a large prime p, replacing each symbol (variable) appearing in the formula by a randomly chosen value from $[0, p)$. The numerical value σ that one obtains is called the signature of f. In this way, if the (possibly large) formula f simplifies to 0, then σ will be 0. If f does not simplify to 0, then σ is "probably" not 0.

It is important to note that for LU factoring, we only need to find a provably non-zero pivot, so that a false positive (σ is 0 but f does not simplify to 0) rarely leads to a problem.

4 Empirical results

We present some timing results. For the benchmarks, we use strategies based on MAPLE's `length` command. As these strategies are heuristics, any reasonable measure of the complexity of an entry is sufficient. The pivoting strategy searches for the element with the largest `length`. The `veiling` strategy depends on the type of matrix. For integer matrices, we veil all integers whose length is greater than 1000, while for polynomial matrices, the threshold is length 30. These constants reflect the underlying constants involved in the arithmetic for such objects. For all benchmarks, three variations are compared:

- our own LU factoring algorithm with veiling and signatures,
- MAPLE's default `LinearAlgebra:-LUDecomposition`,
- a version of `LinearAlgebra:-LUDecomposition` where `Normalizer` has been set to be the identity function and `Testzero` has been set to a version of `testeq`. We had to "patch" MAPLE's `LUDecomposition` to use `Testzero` instead of an explicit call to `Normalizer`, and then had to further "patch" `testeq` to avoid a silly coding mistake that made the code extremely inefficient for large expressions[1].

All tests were first run with a time limit of 300 seconds. Then the first test that timed out at 300 seconds was re-run with a time limit of 1000 seconds, to see if that was sufficient for completion. Further tests in that column were attempted. Furthermore, the sizes of matrices used varies according to the results, to try and focus attention to the sizes where we could gather some meaningful results in (parts of) the three columns. All results are obtained using the TTY version of MAPLE10, running on an 1.8Ghz Intel P4 with 512Megs of memory running Windows XP SP2, and with garbage collection "frequency" set to 20 million bytes used, all results are for dense matrices. In each table, we report the times in seconds, and for the LEM column, the number in parentheses indicates how many[2] distinct labels (ie total number of veiled expressions) were needed by the computation, as an indication of memory requirements. The reason for including the MapleFix

[1]Both of these deficiencies were reported to MAPLESOFT and will hopefully be fixed in later versions of MAPLE

[2]and we use a postfix K or M to mean 10^3 and 10^6 as appropriate

Size n	LEM	MapleFix	Maple
10	.03 (0)	.07	.04
20	.2 (0)	.2	.2
30	.8 (0)	.7	.7
40	2.3 (0)	2.2	2.2
50	6.1 (148)	5.2	5.2
60	12.5 (902)	10.7	10.5
70	17.8 (2788)	19.4	19.2
80	27.6 (5948)	33.8	32.6
90	42.4 (12779)	54.0	52.8
100	56.4 (22396)	83.8	85.8
110	75.4 (36739)	124.7	123

Table 1: Random integer matrices generated using `RandomMatrix(n,n,generator=`$-10^{12}..10^{12}$`)`

column is to really separate out the effect of arithmetic and signature-based zero-testing from the effects of Large Expression Management; MapleFix measures the effect of not doing polynomial arithmetic and using signatures for zero-recognition, and is thus expected to be a middle ground between the other two extremes.

Table 1 shows the result for random matrices over the integers. Only for fairly large matrices (between 90x90 and 100x100) does the cost of arithmetic, due to coefficient growth, become so large that the overhead of veiling becomes worthwhile, as the LEM column shows. Since integer arithmetic is automatic in MAPLE, it is not surprising that the MapleFix column shows times that are the same as the Maple column. Here the veiling strategy really matters: for integers of length 500, veiling introduces so much overhead that for 110x110 matrices, this overhead is still larger than pure arithmetic. For length 2000, no veiling at all occurs.

Table 2 shows the result for random univariate matrices, where the initial polynomials have degree 5 and small integer coefficients. The effect of LEM here is immediately apparent. What is not shown is that MapleFix uses very little memory (both allocated and "used"), while the Maple column involves a huge amount of memory "used", at all sizes, so that computation time was swamped by garbage collection time. Another item to notice is that while the times in the Maple column grow steadily, the ones in the MapleFix column

Size n	LEM	MapleFix	Maple
5	.12 (26)	.06	.53
10	.06 (237)	.07	1.5
15	.18 (872)	.16	9.3
20	.44 (2182)	.30	39.2
25	.87 (4417)	.56	110.4
30	1.9 (7827)	1.87	269.8
35	3.0 (12K)	332	431
40	4.5 (19K)	>1000	845
45	7.8 (28K)	–	>1000
50	9.1 (39K)	–	–

Table 2: Random matrices with univariate entries of degree 5, generated by `RandomMatrix(n,n,generator=(() -> randpoly(x)))`

Size	LEM	MapleFix	Maple
5	.05 (26)	.06	35.3
10	.09 (237)	.09	> 1000
15	.23 (872)	.20	–
20	.49 (2182)	.39	–
25	.99 (4417)	.75	–
30	1.7 (7827)	3.2	–
35	2.8 (12K)	949	–
40	4.2 (19K)	>1000	–
45	6.0 (28K)	–	–
50	8.8 (39K)	–	–

Table 3: Random matrices with trivariate entries, low degree, 8 terms `RandomMatrix(n,n,generator=(() -> randpoly([x,y,z], terms = 8)))`

Size	LEM	MapleFix	Maple
5	.047 (22)	.03	1.56
10	.078 (218)	.08	>1000
15	.20 (858)	.14	–
20	.51 (2163)	.30	–
25	.88 (4393)	.58	–
30	1.7 (7798)	3.8	–
35	2.95 (12K)	>1000	–

Table 4: Fully symbolic matrix: `Matrix(n,n,symbol=m)`

Size	LEM	MapleFix	Maple
5	.031 (26)	xx	0.99
10	.094 (237)	xx	117
15	.22 (872)	xx	> 1000
20	.50 (2182)	xx	–
25	.99 (4417)	xx	–
30	1.7 (7827)	xx	–
35	2.8 (12K)	xx	–

Table 5: Random matrix with entries over $\mathbb{Z}[x, 3^x]$: `RandomMatrix(n,n,generator=(()->eval(randpoly([x,y],terms=8),y=38)))`

are at first consistent with the LEM column, and then experience a massive explosion. Very careful profiling[3] was necessary to unearth the reason for this, and it seems to be somewhat subtle: for both LEM and MapleFix, very small DAGs are created, but for LEM we have full control of these, while for MapleFix, the DAGs are small but the underlying expression tree is enormous. All of Maple's operations on matrix elements first involve the element being *normalized* by the kernel (via the user-inaccessible `simpl` function), and then *evaluated*. While normalization follows the DAG, evaluation in a side-effecting language must follow the expression tree, and thus is extremely expensive. Along with the fact that no information is kept between calls to `testeq`, causes the time to explode for MapleFix for 35x35 (and larger) matrices. Since the veiling strategy used for the last 4 tables is the same, it is not very suprising that the number of veilings is essentially the same. The reason that the all-symbolic is a little lower is because we start with entries of degree 1 and coefficient size 1, and thus these entries do not get veiled immediately. However, one can observe a clear cubic growth in the number of veilings, as expected.

Table 3 shows the result for random trivariate matrices, where the initial polynomials have 8 terms and small integer coefficients. The results here clearly show the effect that multi-variate polynomial arithmetic has on the results. Table 4 shows the results for a matrix with all entries symbolic, further accentuating the results in the trivariate case. Again, MapleFix takes moderate amounts of memory (but a lot of CPU time at larger sizes), while Maple takes huge amounts, causing a lot of swapping and trashing already for 10x10 matrices.

Table 5 shows results for matrices with entries over $\mathbb{Z}[x, 3^x]$. Overall the behaviour is quite similar to bivariate polynomials, however the **xx** in the MapleFix entry indicate a weakness in MAPLE's `testeq` routine, where valid inputs (according to the theory in [16]) return `FAIL` instead. Our signature implementation can handle such an input domain without difficulty.

While we would have liked to present memory results as well, this was much more problematic, as MAPLE does not really provide adequate facilities to achieve this. One could look at **bytes used**, but this merely reflects the memory asked of the system, the vast majority of which is garbage and immediately reclaimed. This does measure the amount of overall memory

[3]Here we used a combination of procedure-level profiling via `CodeTools[Profiling]` and global profiling via `kernelopts(profile=true)`

churn, but does not give an indication of final memory use nor of the true *live set.* **bytes alloc** on the other hand measure the actual amount of system memory allocated. Unfortunately, this number very quickly settles to something a little larger than **gcfreq**, in other words the amount of memory required to trigger another round of garbage collection, for all the tests reported here. This reflects the huge amount of memory used in these computations, but does not reflect the final amount of memory necessary to store the end result. Neither can we rely on MAPLE's **length** command to give an accurate representation of the memory needed for a result because, for some unfathomable reason, **length** returns the expression tree length rather than the DAG length! Thus, for matrices whose results are un-normalized polynomials, we have no easy way to measure their actual size. As a proxy, we can find out the total number of variables introduced by the veiling process.

References

[1] Dongarra, J.J., Duff, I.S., Sorensen, D.C., Van der Vorst, H.A. *Solving Linear Systems on Vector and Shared Memory Computers* SIAM Publications, 1991.

[2] Lengauer T. *Hierarchical Planarity Testing Algorithms* Journal of the Association for Computing Machinery, Vol. 36, No. 3, pp. 474-509, 1989.

[3] Zhou W., Jeffrey D.D., Reid G.J. *Symbolic Preprocessing for the Numerical Simulation of Multibody Dynamic Systems* International workshop on Symbolic-Numeric Computation, Proceedings of SNC, pp 343-354, 2005.

[4] Tinetti F.G., Denham M., Giusti A.D. *Parallel Matrix Multiplication and LU Factorization on Ethernet-Based Clusters*; High Performance Computing, 5th International Symposium, ISHPC, Springer-Verlag Berlin Heidelberg, pp 431-439, 2003.

[5] Dongarra J., Walker D. *Libraries for Linear Algebra* in Sabot F. W. (Ed,), High Performance Computing: Problem Solving with Parallel and Vector Architectures, Addison-Wesley Publishing Company, Inc., pp 93-134, 1995.

[6] Corless R.M. *Essential Maple 7* Springer-Verlag, 2002.

[7] Corless R.M., Jeffrey D.J. , Monagan M.B., Pratibha *Two Perturbation Calculation in Fluid Mechanics Using Large-Expression Management* J. Symbolic Computation, Vol. 23, No. 4, pp 427-443, 1997.

[8] Monagan M.B. Signatures + Abstract Types = Computer Algebra − Intermediate Expression Swell. PhD Thesis, *University of Waterloo*, 1990.

[9] Monagan M.B., Monagan G. *A toolbox for program manipulation and efficient code generation with an application to a problem in computer vision* Proceedings of the 1997 international symposium on Symbolic and algebraic computation, pp 257-264, 1997.

[10] Freeman T., Imizian G., Kaltofen E. *A system for manipulating polynomials given by straightline programs* In proceedings ISSAC'86. Waterloo, 1986.

[11] Díaz A., Kaltofen E. *On computing greatest common divisors with polynomials given by black boxes for their evaluations* In Levelt, A., editor, Proceedings ISSAC'95, Montréal, pp 232-239, 1995.

[12] Budgell P.C., El Maraghy W.H. *Inverse dynamics of the Stanford arm developed with computer symbolic algebra* In Proceedings CSME Mech. Eng. Forum, volum III. Toronto, Canada, 1990.

[13] Zippel R. *Effective Polynomial Computation* Kluwer Academic, 1993.

[14] Monagan M.B. *Gauss: a Parameterized Domain of Computation System with Support for Signature Functions* Proceedings of DISCO'93, Spring-Verlag LNCS, **722**, pp 81-94, 1993.

[15] Steinberg S., Roach P. *Symbolic manipulation and computational fluid dynamics* Journal of Computational Physics, **57**, pp 251-284, 1985.

[16] Gonnet G.H. *Determining Equivalence of Expressions in Random Polynomial Time, Extended Abstract* ACM, pp 334-341, 1984.

[17] Zhou W., Jeffrey D.J., Schost E. *Zero test for Polynomials and Trigonometric Polynomials in Hierarchical Representations* in preparation.

[18] Kaltofen E. *Computing with polynomials given by straight-line programs I: greatest common divisors* Proceedings of the seventeenth annual ACM symposium on Theory of computing, pp 131-142, 1985.

[19] Carette J., Kiselyov O. *Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code.* Generative Programming and Component Engineering, pp 256-274, 2005.