Symbolic Execution Merges Construction, Debugging and Proving

Richard Fateman
Computer Science Division
University of California, Berkeley

December 4, 2002

Abstract

There is naturally an interest in any technology which promises to assist us in producing correct programs. Some efforts attempt to insure correct programs by making their construction simpler. Some efforts are oriented toward increasing the effectiveness of testing to make the programs appear to perform as required. Other efforts are directed to prove the correctness of the resulting program. Symbolic execution, in which symbols instead of numbers are used in what appears to be a numerical program, is an old but to-date still not widely-used technique. It has been available in various forms for decades from the computer algebra community. Symbolic execution has the potential to assist in all these phases: construction, debugging, and proof. We describe how this might work specifically with regard to our own recent experience in the construction of correct linear algebra programs for structured matrices and LU factorization. We show how developing these programs with a computer algebra system, and then converting incrementally to use more efficient forms. Frequent symbolic execution of the algorithms, equivalent to testing over infinite test sets, aids in debugging, while strengthening beliefs that the correctness of results is an algebraic truth rather than an accident.

1 Introduction

Consider a computer program that is intended to compute a mathematical functions such as an inverse of a matrix or a numerical solution to an ordinary differential equation. By a suitable generalization of the operations of a numeric programming system we can allow the inputs and outputs of a function to be symbols or symbolic expressions instead of numbers. Addition of x and y is the expression $x + y$.

The techniques we will discuss pertain primarily to straight-line programs: sequences of asignments, or minor variations on them. However, by allowing

recursion, they become rather more interesting. Our demonstrations are less applicable to programs in which the number of iterations of a loop or the depth of recursion or other branch-decision control-flow depend on particular numerical values of the input. Nevertheless, it is suggestive that if a program works for *all* matrices of size $n = 1, ...10$ that it likely works for larger sizes as well.

The proof for all matrices of a certain dimension – for example that a program that is alleged to compute the determinant of a 3 by 3 matrix is correct– can be tested if it computes the determinant of a symbolic 3 by 3 matrix of 9 independent symbols, and abides by certain other criteria described in the next section. The result may not be in the same format, and as is well known in the specific case of the determinant, numerous alternative arrangements of the answer are possible. A computer algebra system can however compute a correct determinant or test to see if the difference of two symbolic results is identically zero. Thus the zero-ness of the expression below, the difference of two determinant calculations. is easily computed.

$$(a\,e\,i - b\,d\,i - a\,f\,h + c\,d\,h + b\,f\,g - c\,e\,g) - a\,(e\,i - f\,h) - b\,(d\,i - f\,g) + c\,(d\,h - e\,g)$$

In the course of this paper we essentially prove that a certain computer program computes an LU factorization (actually a whole family of programs), and that the programs work for several different representations.

One of the major tools we will use to start is a purely functional approach to programming. This means that each procedure is equivalent to a mathematical function: it computes a function of its input and returns a value. In particular it does not change its input in any way. Furthermore, each distinct variable can be assigned a value only once. To programmers unused to this approach it may seem impossible to write programs this way at all.

2 Criteria for validity

The assumptions used in our discussion may not always be met in practice. Thus developing foolproof programs must be done within a framework recognizing that some issues cannot be resolved by the idealized arithmetic of computer algebra systems when they are applied to approximate floating-point arithmetic. (That is to say, the usual cautions must still be observed, even if the arithmetic, done perfectly, is correct).

One set of assumptions could be:

1. All floating-point operations performed during the execution of the program are closed within the representable floating-point numbers. If there is a division, it must not be by zero. If there is an exponentiation, it must not be 0^0.

2. All integer operations in the program are closed within the representable integer numbers. If there is a division, it must not be by zero. If there is an exponentiation, it must not be 0^0.

3. No floating-point or integer overflows may occur. No floating-point underflows may occur.

4. Floating point errors caused by truncation or roundoff are assumed to be insignificant in the sense that they do not affect the correctness of the program.

These criteria mean that algorithms which depend on iterative convergent behavior cannot be directly treated by symbolic execution, although rather subtle behavior related to convergence can sometimes be observed in symbolic execution.

It is ordinarily NOT possible to generate a proof by induction automatically. A matrix proof would likely work only for specific size matrices, say 2 by 2, 3 by 3, 5 by 5. It is not (usually) possible to compute an indefinite number of terms, say n^2. The matrix size n must be defined.

3 LU factorization

We came upon this problem because of a talk given by Fred Gustavson [1] on a technique to preserve locality in LU factorization of a dense matrix. The improved performance caused by cache coherency was significant. Our issue here is to figure out how to compose, debug, and prove correct variants of LU factorization. We will not dwell upon the reason for LU factorization to be interesting (It allows for the faster repeated solving of matrix-vector equations $Ax = b$ for varying right-hand sides if A is factored), nor on the many many variations of the algorithm (based on parallel computation, sparse matrices, matrices too large to fit in main memory.) In fact we will write algorithms that are almost entirely recursive in nature, an idea that Gustavson shows provides a substantial measure of memory locality using his data rearrangements. (Actually, our programs provide recursive subdivision down to 1 by 1 matrices; in reality one would generally use recursion down to a size which fits conveniently into cache memory and then use other machine-optimized block programs.)

The idea is to factor an m by m matrix A into a product of matrices L and U. L is a unit lower triangular matrix: that is, all entries above the diagonal are 0, all entries on the diagonal are 1. The algorithm must find the entries below the diagonal. The matrix U is upper triangular: all entries below the diagonal are 0. The algorithm must find the all remaining entries in U.

Here is the LU factorization for a 2 by 2 matrix ($a \neq 0$):

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{c}{a} & 1 \end{pmatrix} \cdot \begin{pmatrix} a & b \\ 0 & d - \frac{bc}{a} \end{pmatrix}$$

In fact we can reduce the problem using any size blocks between 1 and $m-1$. If the matrix A is square and of even order, we could divide the matrices into four equal square blocks, as suggested by Toledo [2].

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$

Note that L_{11} and L_{22} are lower unit triangular and that U_{11} and U_{22} are upper triangular. Also, if A is square, the blocks on the diagonal of L and U are also square.

1. Recursively solve the problem $A_{11} = L_{11} \cdot U_{11}$ giving values for entries in L_{11} and U_{11}.

2. Recursively solve the problem $A_{12} = L_{11} \cdot U_{12}$ giving values for entries in L_{11} and U_{11}. Alternatively, since L_{11} is known from the previous step, at the cost of inverting L_{11} we can compute $U_{12} = L_{11}^{-1} \cdot A_{12}$. (Or more economically, we could use a procedure to solve for U_{12} without explicitly inverting L_{11}.)

3. Recursively solve the problem $A_{21} = L_{21} \cdot U_{11}$ giving values for entries in L_{21} and U_{11}. Alternatively, since U_{11} is known from a previous step, at the cost of inverting U_{11} we can compute $L_{21} = A_{21} \cdot U_{11}^{-1}$. (Or more economically, we could use a procedure to solve for L_{21} without explicitly inverting U_{11}.)

4. Since $A_{22} = L_{21} \cdot U_{12} + L_{22} \cdot U_{22}$ we can recursively solve the $A' = LU$ factorization problem $A_{22} - L_{21} \cdot U_{12} = A' = L_{22} \cdot U_{22}$ to give the values for entries in L_{22} and U_{22}.

5. Return all the computed entries and we are done.

4 Inverse of a lower unit triangular matrix L

Let's treat just one of the subproblems in more detail.

4.1 Example

An $n \times n$ lower unit triangular matrix has all zeros above the (main) diagonal, ones on the diagonal, and arbitrary values below the diagonal. Here's a 4×4 example

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ a21 & 1 & 0 & 0 \\ a31 & a32 & 1 & 0 \\ a41 & a42 & a43 & 1 \end{pmatrix}$$

We have used symbols such as a43 to mean arbitrary values, numerical or symbolic as suits the problem. Naturally it is possible to save memory (and the speed of accessing memory) if the non-trivial values are stored sequentially, and the trivial values of 1 and 0 are not stored at all. That is, they are implicit in the representation. Thus this particular matrix has only 6 interesting values, and

they can, for example, be arranged in a vector of 3 vectors: {a21, a31, a41}, {a32, a42} and {a43}. Computing any function f of M should involve accessing only these 6 values. If f results in a lower unit triangular (LUT) answer, the answer should involve only six values as well.

4.2 The Form of the Inverse

Some introspection or some calculation easily set up in a computer algebra system, or even a numeric matrix system such as Matlab or its free clones, quickly reveals.

$$M^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a21 & 1 & 0 & 0 \\ a21\,a32 - a31 & -a32 & 1 & 0 \\ (a31 - a21\,a32)\,a43 + a21\,a42 - a41 & a32\,a43 - a42 & -a43 & 1 \end{pmatrix}$$

We can see from this that the inverse of ANY 4×4 LUT is LUT, and it is not (say) an accident of the random numbers we have chosen, or of the floating-point vagaries of the computation. We can guess and then prove that the inverse of a LUT matrix is LUT, perhaps this way:

We can observe that an LUT matrix can be blocked into two (not necessarily equal) LUT square matrices on the diagonal, plus a full matrix below, and a zero matrix above.

$$\begin{pmatrix} L1 & 0 \\ A & L2 \end{pmatrix}$$

If we compute the inverse in Macsyma, we get the rather unconvincing

$$\begin{pmatrix} \frac{1}{L1} & 0 \\ -\frac{A}{L1\,L2} & \frac{1}{L2} \end{pmatrix}$$

clearly an error since the computer algebra system doesn't know that L1, L2 and A are non-commuting block matrices. However some thought tells us that

$$\begin{pmatrix} L1^{-1} & 0 \\ -L2^{-1} \cdot A \cdot L1^{-1} & L1^{-1} \end{pmatrix}$$

is a suitable rendering, and that it gives us a way of nicely computing the inverse of an LUT matrix recursively.

It appears that we are almost done if we can figure out how to

1. Break apart a matrix into blocks.

2. Terminate the recursion: This is easy. by checking for a matrix of size 1, we can halt the subdivision and write down the inverses.

3. Compute the product of three matrices to fill in the lower left corner.

4. Assemble a matrix from blocks.

What we must compute is

$$-\text{L2}^{-1} \cdot \text{A} \cdot \text{L1}^{-1}$$

which could be done by reference to the two already-computed inverses and the remaining part of the matrix. It is a product of an LUT times a full matrix times another LUT. As can be seen by the expression below, instead of requiring the full 16 (2 times 2^3) multiplies of submatrices, this requires only 12.

$$\begin{pmatrix} a & 0 \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} \cdot \begin{pmatrix} i & 0 \\ k & l \end{pmatrix} = \begin{pmatrix} a \cdot (f \cdot k + e \cdot i) & a \cdot f \cdot l \\ d \cdot (h \cdot k + g \cdot i) + c \cdot (f \cdot k + e \cdot i) & (d \cdot h + c \cdot f) \cdot l \end{pmatrix}$$

Note that we are taking advantage of the fact that $(f \cdot k + e \cdot i)$ appears twice. As subproblems we must recursively be able to pre- and post-multiply ordinary matrices by LUT matrices (here, a, d, i, l).

$$\begin{pmatrix} a & 0 \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a \cdot e & a \cdot f \\ d \cdot g + c \cdot e & d \cdot h + c \cdot f \end{pmatrix}$$

$$\begin{pmatrix} e & f \\ g & h \end{pmatrix} \cdot \begin{pmatrix} i & 0 \\ k & l \end{pmatrix} = \begin{pmatrix} f \cdot k + e \cdot i & f \cdot l \\ h \cdot k + g \cdot i & h \cdot l \end{pmatrix}$$

We also need two programs to manipulate blocks. The first partitions a matrix into blocks, and the second reassembles the blocks. If we are careful in designing data structures, (and initially we will *not* be especially careful), then one might arrange for matrix partitioning by (in effect) pointing to some place in the midst of a matrix and say "The new matrix starts right here. Index starting from this spot." An ordinary two-dimensional array layout does not support this operation without recopying or some fancy indexing footwork, but a one-dimensional array usually does.

What does this look like in a particular symbolic programming language? Here we use Macsyma.

```
/Macsyma programs for inverse of LUT matrices*/

/* First, a program that makes it easy to construct compactly
printable LUT matrices with entries aij (for testing)*/

(a[i,j]:= if i=j then 1 else if j>i then 0 else concat(a,i,j),

/* LUI recursively computes the inverse of an LUT matrix */

scalarmatrixp:false, /*a flag in Macsyma controlling conversion
of matrix to scalar */

lui(m):=block([h:length(m),p,l1,l2,g],
          if h=1 then m else
```

```
        (g:ceiling(h/2), /* break about in half */
         p:partmat(m,g,g),
         l1:lui(p[1]),
         l2:lui(p[4]),
         tget(l1,-l2.p[2].l1,p[3],l2)))),
```

This is really all the program that is needed except for the partitioning and reassembling. `Partmat` creates a list of 4 matrices by partitioning off the top left i,j rows and columns. And `tget` take 4 matrices in the same order as returned by `partmat` and plunks them together into one matrix.

```
partmat(m,i,j):=block([hi:length(m),wide:length(m[1])],
  [submat(m,1..i,1..j),      submat(m,i+1..hi,1..j),
   submat(m,1..i,j+1..wide),submat(m,i+1..hi,j+1..wide)]),

tget(m1,m2,m3,m4) := addrow(addcol(m1,m3),addcol(m2,m4)))
```

Note that we have written this program in a purely functional fashion: variables are assigned values only once. There are no loops (although LUI is recursive). Does this purely functional program work?

```
 ratsimp( a6:genmatrix(a,6,6) . lui(a6));
```

produces a 6 by 6 identity matrix. This is actually proof that LUI works for *any* non-singular 6 by 6 LUT matrix. Proving that it works for matrices of both odd and even size, including sizes 1, 2, 3, 4 is fairly convincing evidence that it works generally. Another version of a proof is to compare the result of the Macsyma matrix inversion program to this one. They produce the same answers though not arranged in identical format.

4.3   More compact representation

Assume that the LUT matrices we are dealing with should not be represented in full. In particular, an n by n LUT matrix has only $n(n-1)/2$ entries that must be stored, since the others form a predictable pattern of 0 and 1. In order to pursue this we must write each of the programs to fit the new data format, including the matrix multiplication operation (the operator ".") we have used casually in the program above.

```
/* Convert ordinary dense matrix to LUT representation.
   And the reverse. Provide for operations. Keep the
   programs short if not efficient.

 The matrix is the same as

   1 0 0
   a 1 0      lut ([a,b],[c]).
   b c 1
```

Note that `lut` is merely a tag or header for a data structure consisting of a number of lists. There is no procedure associated with this tag.

```
(lutp(r):=operatorp(r,lut),  /* is something an lut? */

/* implement lut-to-matrix conversion and operation of '.'
the latter by adding patterns to the simplifier. */

matchdeclare([l1,l2],lutp, any,true),
tellsimp(l1 . any, lut2mat(l1) . any),
tellsimp(any . l1, any . lut2mat(l1)),
tellsimp(l1 . l2, mat2lut(lut2mat(l1) . lut2mat(l2))),

declare(lut,nary),  /* lut(a,lut(b)) becomes lut(a,b) */

mat2lut(m):= block([d:length(m)], if (d<=1) then lut()
                else lut(submat(m,2..d,1..1),
                          mat2lut(submat(m,2..d,2..d)))),

lut2mat(r):=block([d:length(r)], if d=0 then matrix([1])
                else addcol (addrow(matrix([1]), first(r)),
                              addrow(zeromatrix(1,d),
                                      lut2mat(rest(r))))))
)
```

Here is how the two forms look:

$$
\begin{pmatrix} 1 & 0 & 0 & 0 \\ a21 & 1 & 0 & 0 \\ a31 & a32 & 1 & 0 \\ a41 & a42 & a43 & 1 \end{pmatrix} \text{ encodes as lut} \left(\begin{pmatrix} a21 \\ a31 \\ a41 \end{pmatrix}, \begin{pmatrix} a32 \\ a42 \end{pmatrix}, (a43) \right)
$$

We could make LUI work for LUT forms by converting the input to ordinary matrix form and using the same old program, but that would hardly be respectable. Especially since at least one form of the program, ripping the left column off and doing a matrix-by-vector multiplication, is much neater.

Here is a program working on lut representation.

```
luilut(a):= if length(a)=0 then lut()
        else block([z:luilut(rest(a))],
                lut(z.(-first(a)),z)),
)
```

Unfortunately, that program still uses the ordinary "." matrix multiply, and in the process computes a LUT matrix by vector product by expanding the matrix and using the built-in matrix product. This is hardly aesthically pleasing, so we look for a neat way to write this out:

8

```
( /* matrix . vector where matrix is in LUT form, vector is a column.
  the result is a normal (column) vector. */

dotlutv(l,v) := If l = lut() then v else
   addrow(matrix(first(v)),
        v[1,1]*first(l)+dotlutv(rest(l),rest(v))),

/* and now we rewrite lu inverse in lut form as this: */

luilut(a):= if length(a)=0 then lut()
          else block([z:luilut(rest(a))],
               lut(dotlutv (z,-first(a)),z))
)
```

Are we still on track? Consider the general 4 by 4 LUT matrix

$$\mathrm{lut}\left(\begin{pmatrix} \mathrm{a21} \\ \mathrm{a31} \\ \mathrm{a41} \end{pmatrix}, \begin{pmatrix} \mathrm{a32} \\ \mathrm{a42} \end{pmatrix}, (\,\mathrm{a43}\,)\right).$$

Multiplying this by its inverse computed by the program above,

$$\mathrm{lut}\left(\begin{pmatrix} -\mathrm{a21} \\ \mathrm{a21\,a32} - \mathrm{a31} \\ -\mathrm{a21}\,(\mathrm{a32\,a43} - \mathrm{a42}) + \mathrm{a31\,a43} - \mathrm{a41} \end{pmatrix}, \begin{pmatrix} -\mathrm{a32} \\ \mathrm{a32\,a43} - \mathrm{a42} \end{pmatrix}, (\,-\mathrm{a43}\,)\right)$$

produces

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\mathrm{a21}\,(\mathrm{a32\,a43} - \mathrm{a42}) + (\mathrm{a21\,a32} - \mathrm{a31})\,\mathrm{a43} + \mathrm{a31\,a43} - \mathrm{a21\,a42} & 0 & 0 & 1 \end{pmatrix}$$

which is easily reduced, by issuing a simplification command, to the identity matrix.

The next direction is to depart from our functional approach, and instead of returning a newly formed answer, we take any piece of the input array M and replace entries as we go. At the termination of the algorithm we have destroyed M by replacing it with its inverse.

A common technique would be to pass in as an extra argument to each function the location in which the result should be placed. If we are correct in believing we can overlay existing data with the results, then the target can be the same as the input. The usual method for such in-out parameters in Fortran is to just provide comments about the usage and plow ahead. In a functional programming language we have to break the pattern in some way, either by passing the names of items to be changed into a program, or using the destructive operations (exceptions to the pure-function orientation) available.

What we really need is just this:

```
 /* matrix . vector where matrix is in LUT form, vector
    v is a column.  The result is accumulated in a vector
    ans, which if it is supplied should be a copy of v. */

(dx(l,v):= /* testing program */
block([ans:copymatrix(v)],dotlutvx(l,v,ans)),

dotlutvx(l,v, ans):=
(if l # lut() then
     (for i from 2 thru length(v) do
             ans[i] : ans[i] + v[1] * first(l)[i - 1],
             dotlutvx(rest(l), rest(v))),
             ans))
```

Note that Macsyma (and Lisp) really don't have call by reference thus alteration
of a parameter is usually done by passing in an array or list and changing
"destructively" the elements in that value.

5   LU factorization

There is a problem composing an LU factorization procedure in a purely func-
tional notation along the lines that we have used in the program above. Namely,
a conventional function return a single value, and the LU factorization result of
a matrix A consists of two matrices. The traditional Fortran approach to this
is to pack the two matrices together into a matrix the same size as A, and in
fact to place this result on top of A, destroying its former value.

We would like to first develop the program as though it were functional. If
it seems appropriate, then as a storage optimization, we can consider how to
destroy A and re-use its storage.

One way to return two items in languages allowing more flexible than Fortran
is to return some kind of object constituting this collection. For example, return
a list or a pair, [L,U]. (Common Lisp allows for multiple-value returns, but this
is not tremendously more convenient than pairing-up). By contrast, Fortran
programmers usually destroy some of the input arguments or store "extra"
return values in globally accessible places. These Fortran tricks make the use
of recursive programs difficult.

Here's a program that works for any size square LU-factorable matrix (it
need not be of even dimension) on a full matrix A and returns full matrices for
L and U.

```
luf(m):=block([h:length(m), g, m11,m12,m21,m22,
               L11,L21,L22,    u11,u12,u22],
               if h=1 then [matrix([1]), m] else
             (g:ceiling(h/2), /* break in half */
              [m11,m21,m12,m22]:partmat(m,g,g),
```

```
    [L11,u11]: luf(m11),
          u12: invert(L11).m12,
          L21: m21.invert(u11),
    [L22,u22]: luf(m22-L21.u12),
    [tget(L11,L21,zeromatrix(g,h-g),L22), /*L*/
     tget(u11,zeromatrix(h-g,g),u12,u22) /*U */]))
```

We can write programs that compute u12 and L21 without actually computing those inverses, removing the need for storage of these quantities. For example, instead of `m21.invert(u11` we can use muinv(m21,u11) which is defined below:

```
/*compute M.invert(U), U is upper triangular. Similar
programs for invert(L) */

muinv(m,u):=block([d:length(u), g, m11,m12,m21,m22,
             h11,h12,h21,h22,   u11,u12,u21, u22],
              if d=1 then  m/part(u,1,1) else
            (g:ceiling(d/2), /* break in half */
             [m11,m21,m12,m22]:partmat(m,g,g),
             [u11,u21,u12,u22]:partmat(u,g,g),
             h11: muinv(m11,u11),
             h21: muinv(m21,u11),
             h12: muinv(m12-h11.u12,u22),
             h22: muinv(m22-h21.u12,u22),
             tget(h11,h21,h12,h22)))
```

We can make the LUF program slightly shorter by just splitting off the first row/column, not trying to subdivide into roughly-equal sections. Knowing that the top row and left column are being peeled off leads to other simplifications.

```
luf1(m):=block([h:length(m), m11,m12,m21,m22,
              L21,L22,      u11,u12,u22],
              if h=1 then [matrix([1]), m] else
            ([m11,m21,m12,m22]: partmat(m,1,1),
                   [u11,u12]: [m11, m12],
                        L21: m21.invert(u11),
                  [L22,u22]: luf1(m22-L21.u12),
            [tget(matrix([1]),L21,zeromatrix(1,h-1),L22), /*L*/
             tget(u11,zeromatrix(h-1,1),u12,u22) /*U */]))
```

This is probably about as small a program as one can get for the LU factorization problem (though we have not addressed the complications that might be necessitated by pivoting).

The next step is to address storage issues, critical especially for large matrices which must be fit into limited cache-size high-speed memory. There are two

complementary approaches, namely maintaing more coherent local use of caches, and also reducing the total storage. We can reduce storage if we do not represent the L variables as full matrices, as suggested previously, and similarly for the U matrices. We propose to represent the upper triangular matrices (UT) as a structure which collects some rows, omitting the zero entries:

$$\begin{pmatrix} u11 & u12 & u13 \\ 0 & u22 & u23 \\ 0 & 0 & u33 \end{pmatrix} = \mathrm{ut}\left(\{u11, u12, u13\}, \{u22, u23\}, \{u33\}\right)$$

We would like to compute with the UT representation directly. With minor variations, we can re-use most of the program framework from LUT matrices to get

```
(utp(r):=operatorp(r,ut),  /* is something an ut? */
u[i,j]:= if  j<i then 0 else concat(u,i,j),

/* implement ut-to-matrix conversion and operation of '.'
the latter by adding patterns to the simplifier. */

matchdeclare([u1,u2],utp, any,true),
tellsimp(u1 . any, ut2mat(u1) . any),
tellsimp(any . u1, any . ut2mat(u1)),
tellsimp(u1 . u2, mat2ut(ut2mat(u1) . ut2mat(u2))),
declare(ut,nary),  /* ut(a,ut(b)) becomes ut(a,b) */

mat2ut(m):= block([d:length(m)], if (d=0) then ut()
                else ut(first(m),
                         mat2ut(submat(m,2..d,2..d)))),

ut2mat(r):=block([d:length(r)], if d=1 then matrix(first(r))
            else addrow (matrix(first(r)),
                         addcol(zeromatrix(d-1,1),
                        ut2mat(rest(r)))))
)
```

At this point we can begin experimenting with more compact representations of both lower and upper matrices. In fact we can combine the two parts of a matrix into a UT (upper triangular) and LUT (lower unit triangular) section. Except in the LU factorization routines, we will not claim the diagonal of an LUT consists of ones. We can just deal with the two parts, arranged as collections of vectors, as a full triangular (ft) matrix: Thus

$$\begin{pmatrix} q_{1,1} & q_{1,2} & q_{1,3} \\ q_{2,1} & q_{2,2} & q_{2,3} \\ q_{3,1} & q_{3,2} & q_{3,3} \end{pmatrix} = \mathrm{ft}\left(4,\ \mathrm{ut}\left(\{q_{1,1}, q_{1,2}, q_{1,3}\}, \{q_{2,2}, q_{2,3}\}, \{q_{3,3}\}\right), \mathrm{lut}\left(\begin{pmatrix} q_{2,1} \\ q_{3,1} \end{pmatrix}, \begin{pmatrix} q_{3,2} \end{pmatrix}\right)\right)$$

Here's the basic idea for the FT matrices. We will keep track of the dimension, and not have it implicit in the lengths of lists.

To split a square FT of dimension divisible by 2 into equal-sized partitions, there is relatively little work compared to copying over the whole matrix. For specificity, consider a 4 by 4 matrix. The upper left 2 by 2 corner of an R=FT(4, U, L) is simply the same form but with a 2 replacing the 4. We simply will not look at the trailing values in U and L. The lower right 2 by 2 corner is FT(2, rest(U,2), rest(L,2)). If U and L are set up as lists, this may take linear time in the dimension of R. If U and L are set up as vectors, which we would expect for anyone concerned with speed, this should take constant (and very small) time.

The lower-left corner must be extracted from L, creating a new vector of length 2 which will point to the 2nd element of column 1 and the first element of column 2 (that's all). This is not an FT matrix any more, but a full matrix stored column by column. (FC)

The upper-right corner must be extracted from U, creating a new vector of length 2 which will point to the 3rd element of row 1 and the 2nd element of row 2. This is not an FT matrix any more, but a full matrix stored row by row. (FR)

Multiplying two square FT matrices A and B together can be decomposed recursively into multiplying 4 block matrices in A and B. This (in the usual approach) requires 8 matrix multiplications, of the follow types and numbers:

```
FT by FT: 2
FR by FC: 1
FC by FR: 1
FT by FR: 1
FT by FC: 1
FR by FT: 1
FC by FT: 1
```

This becomes somewhat tedious to program, but not difficult. A slight motification to this matrix-matrix multiply which has a certain appeal recursively is to make the upper-left corner be a single (scalar) quantity, and do the matrix multiply by peeling off only the first row and left-most column of each input.

Then we addressing the matrix-matrix multiply of

$$\begin{pmatrix} a & B_r \\ C_c & D_m \end{pmatrix} \cdot \begin{pmatrix} e & F_r \\ G_c & H_m \end{pmatrix}$$

where a and e are scalars, B_r and F_r are rows, C_c and G_c are columns, and D_m and H_m are full matrices of diminished (by 1) dimension.

The resulting block matrix looks like this:

$$R = \begin{pmatrix} ae + B_r \cdot G_c & a \cdot F_r + a \cdot H_m \\ C_c \cdot e + D_m \cdot G_c & C_c \cdot F_r + D_m \cdot H_m \end{pmatrix}$$

We have reduced the complexity of the computations in each of the quadrants except one. We are doing (vector by scalar) or (vector by vector) or (vector by

matrix) operations except in the lower left corner where there is the recursive matrix by matrix multiplication. Note that $C_c \cdot F_r + D_m \cdot H_m$ should not be computed by multiplying $C_c \cdot F_r$ into a matrix and then adding. Rather each entry result from $D \cdot H$ should be modified by adding the easily computed additional term: the product of one element of C and one from F. This leads us to a modification of the original specification. Instead of writing a program to compute a matrix product, why not specify `f(A,B,row,col)` where the result is the computation `col.row+A.B` where `A,B` are n by n matrices, and `col` and `row` are n by 1 and 1 by n respectively (as should be clear, each of these latter objects would simply occupy n adjacent locations in memory). A further step is to write out the the computation as adding an indefinite number of such cross products to a matrix which is, *in extremis*, empty, and recursively building up the result.

The point of this exercise is to write out programs (and prove them correct) where we make this problem smaller and smaller by recursion, but we might do this only until H, D and R all fit into fast memory. But what if significant parts of the computation don't? For example, imagine a matrix for which no single row or column fits in cache. Our expectation is that additional patterns of subdivision using sub-vectors can be constructed, with the hope of (for example) nailing down most of the cache and running the rest of the data through the remaining part of the cache, inevitably causing cache misses: a larger cache devoted to this would not help. Such sophisticated control is generally not provided to a programmer in a higher-level language, if indeed it is even made available to the assembly-language programmer.

Even so, if we can formulate experiments in shifting the program forms about so we can retain one of them, say D in fast memory and compute all the columns of R moving those results out of cache; then we can retain H in fast memory and compute the rows of R: it would be advantageous if we can see that they are debugged symbolically.

More examples along these lines are in the works.

6   Conclusions

It's hard to get programs right. Clear confirmation of some sense of correctness through symbolic execution can be helpful. Additional modifications of programs in clever ways to take advantage of rearranged data structures, memory access patterns or other complications can speed up programs, but again a clear confirmation that the computation is still algebraically the same, is valuable. Symbolic tools can help.

References

[1] F. Gustavson, New Generalized Data Structure for Matrices Leading to a Variety of High Performance Algorithms IFIP WG2.5 presentation Octo-

ber, 2000, Ottawa, Canada (to be published by Kluwer).

[2] Sivan Toledeo, "Locality of Reference in LU Decomposition with Partial Pivoting," SIAM J. Matrix Anal. App.. vol 18 no 4 (Oct, 1997) pp.1065–1081.