

Look Left, Look Right, Look Left Again: An Application of Fractal Symbolic Analysis to Linear Algebra Code Restructuring

Vijay Menon and Keshav Pingali

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. Fractal symbolic analysis is a symbolic analysis technique for verifying the legality of program transformations. It is strictly more powerful than dependence analysis; for example, it can be used to verify the legality of blocking LU factorization with pivoting, a task for which dependence analysis is inadequate. In this paper, we show how fractal symbolic analysis can be used to convert between left-looking and right-looking versions of three kernels of central importance in computational science: triangular solve, Cholesky factorization, and LU factorization with pivoting.

1 Introduction

Many computational science applications require the solution of systems of linear equations. These systems can be written in the form $Ax = b$ where A is a matrix, b is a vector of known values, and x is the vector of unknowns. Algorithms for solving such systems can be divided into *iterative* methods and *direct* methods. Iterative methods such as conjugate gradient and GMRES repeatedly refine an initial approximation to the solution of the linear system until they obtain an approximation which is close to the actual solution. Direct methods for solving linear systems factorize the matrix A into a product of an upper triangular matrix and a lower triangular matrix, and then find x by solving the two triangular systems. If the matrix is symmetric and positive-definite, Cholesky factorization is usually used to find the two triangular factors; otherwise, LU with partial pivoting is used.

In this paper, we will focus on direct methods for solving linear systems. Therefore, the kernels of interest to us are (i) Cholesky factorization, (ii) LU factorization with partial pivoting, and (iii) triangular solve.

Each of these kernels has been coded in a variety of ways in the literature. The most important variations are classified as *right-looking* or *eager* codes, and *left-looking* or *lazy* codes.

- *Right-looking codes:* In matrix factorization, the matrix is traversed by columns from left to right. At each step, computations are performed on the *current column*, and then updates to columns to the right of that current column

are performed immediately. Because updates are performed right away, right-looking formulations are sometimes called *eager* formulations.

Figure 1(a) shows right-looking Cholesky factorization. In this code, the current column is indexed by k , and the columns to the right of the current column, which are updated eagerly after the current column has been computed, are indexed by j . Figure 2(a) shows right-looking LU factorization with partial pivoting. In this kernel, partial pivoting is similar to an update operation; when elements of the current column are swapped, swaps are performed on columns to the right of the current column as well. Finally, Figure 3(a) shows right-looking triangular solve. In this code, the outer loop computes each unknown $x(k)$ successively, and the contributions of this unknown to the remaining equations are immediately subtracted from these equations.

- *Left-looking codes*: As in right-looking codes, the matrix is traversed by columns from left to right. At each step, updates to the current column from previous columns are performed, and then computations are performed on the current column. Because updates to a column from previous columns are performed as late as possible, these versions are also known as *lazy* formulations.

Figure 1(b) shows left-looking Cholesky factorization. In this code, the current column is indexed by j , and the columns to the left of the current column are indexed by k . Figure 2(b) shows left-looking LU factorization with partial pivoting, while Figure 3(b) shows left-looking triangular solve.

Neither right- nor left-looking forms should be viewed as canonical. For example, Golub and van Loan’s textbook on matrix computations [3] presents triangular solve and Cholesky factorization using the left-looking or lazy version, and LU factorization with pivoting using the right-looking or eager version. Moreover, neither the left-looking nor the right-looking versions of these codes perform uniformly better, as Figure 4 demonstrates¹. The storage layout of a matrix in memory and the way in which it is distributed across multiple processors may lead to a preference for one or the other of these formulations. Therefore, it is important for compilers to be able to convert freely between left- and right-looking versions of these algorithms, using general-purpose program transformation technology.

In Section 2, we show that conversion between left-looking and right-looking versions of the three kernels can be viewed as a special case of a general transformation that we call *right-left interchange*. The problem is to prove that this transformation can be legally applied to the three kernels of interest.

The most commonly used technique for proving legality of transformations is *dependence analysis* [9]. Dependence analysis computes a partial order between statements, based on the sets of locations touched by those statements. If two statements touch the same memory location and at least one statement writes to

¹ These numbers were obtained on a 300 MHz SGI Octane with a 2MB L2 cache, and an R12K processor. All compiled code was generated using the SGI MIPSpro f77 compiler with flags: -O3 -n32 -mips4.

```

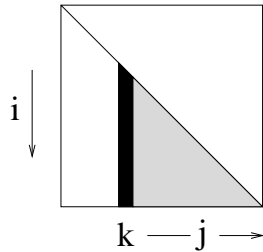
do k = 1,N
B1(k) :// Take square root
      // and scale current column
      A(k,k) = sqrt(A(k,k))
      do i = k+1,N
        A(i,k) = A(i,k)/A(k,k)

      // Update from current column
      // to columns to right
      do j = k+1,N
        B2(k, j) : do i = j,N
          A(i,j) = A(i,j)-A(i,k)*A(j,k)

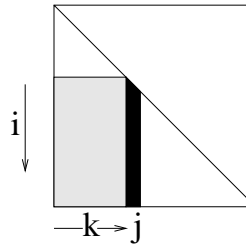
do j = 1,N
      // Update from columns
      // to left to current column
      do k = 1,j-1
        B2(k, j) : do i = j,N
          A(i,j) = A(i,j)-A(i,k)*A(j,k)

      // Scale current column
      B1(j) : A(j,j) = sqrt(A(j,j))
      do i = j+1,N
        A(i,j) = A(i,j)/A(j,j)

```



(a) Right-looking Cholesky



(b) Left-looking Cholesky

Fig. 1. Cholesky Factorization

that location, the compiler assumes that a dependence exists between them and that they cannot be reordered without violating the semantics of the program. More precisely, this analysis is performed on *statement instances* (executions of statements within loops for particular index values of loops enclosing those statements), and a loop transformation is assumed to be legal only if it respects all dependences between instances of statements contained within it. Dependence analysis provides a sufficient but not necessary condition for legality of a transformation.

In Section 3, we show that dependence analysis is adequate to prove the legality of conversion between left-looking and right-looking forms of triangular solve and Cholesky factorization. Unfortunately, dependence analysis is not adequate for LU factorization with pivoting.

In principle, this inadequacy of dependence analysis can be addressed by using *symbolic program analysis*. Symbolic analysis compares two programs for equality by deriving symbolic expressions for the outputs of these programs as functions of their inputs, and then attempting to prove that these expressions are equal. This analysis technique is very powerful, and in principle, it can be used to prove equality of even different algorithms such as heap-sort and merge-sort. In practice though, symbolic analysis is undecidable or intractable for all but the simplest codes. In particular, we do not know any practical way of performing symbolic analysis of LU factorization.

```

do k = 1, N
  // Pick the pivot
  B1.a(k) :
    p(k) = k
  B1.b(k) :
    do i = k+1, N
      if abs(A(i,k)) > abs(A(p(k),k))
        p(k) = i

  // Swap rows
  B1.c(k) :
    do j = 1, N
      tmp = A(k,j)
      A(k,j) = A(p(k),j)
      A(p(k),j) = tmp

  // Scale current column
  B1.d(k) :
    do i = k+1, N
      A(i,k) = A(i,k) / A(k,k)

  // Update from current column
  // to columns to right
  do j = k+1, N
    B2(k,j) :
      do i = k+1, N
        A(i,j) = A(i,j) - A(i,k)*A(k,j)
enddo

do j = 1, N
  // Swap rows from left
  do k = 1, j-1
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp

  // Update from columns to left
  // to current column
  do k = 1, j-1
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)
    enddo

  // Pick the pivot
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i
    enddo

  // Swap rows to the left
  do k = 1, j
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp
  enddo

  // Scale current column
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)
  enddo
enddo

```

(a) Right-looking LU

(b) Left-looking LU

Fig. 2. LU Factorization with Partial Pivoting

```

do k = 1, N
  // Compute current unknown
  B1(k) : x(k) = x(k)/A(k,k)

  // Update from current unknown
  // to later unknowns
  do j = k+1, N
    B2(k,j) : x(j) = x(j) - A(j,k)*x(k)
  enddo
enddo

do j = 1, N
  // Update from earlier unknowns
  // to current unknown
  do k = 1, j-1
    B2(k,j) : x(j) = x(j) - A(j,k)*x(k)
  enddo

  // Compute current unknown
  B1(j) : x(j) = x(j)/A(j,j)
enddo

```

(a) Right-looking Triangular Solve

(b) Left-looking Triangular Solve

Fig. 3. Lower Triangular Solve

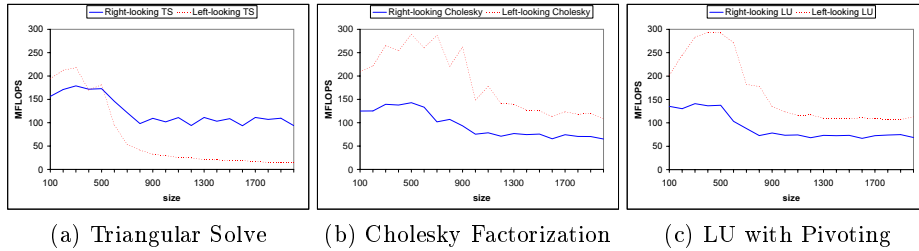


Fig. 4. Performance of Left- and Right-looking Codes

To bridge this gap between dependence analysis and symbolic analysis, we have developed a new analysis technique called *fractal symbolic analysis* [6]. Fractal symbolic analysis is a form of symbolic analysis that can be used to prove equality of programs *when the two programs are related by a restructuring transformation*. If the two programs are “simple enough”, symbolic analysis is used directly to verify their equivalence. If comparing the two programs symbolically is too complicated, fractal symbolic analysis simplifies the two programs, *ensuring that equality of the simplified programs implies equality of the original programs*. The restructuring transformation that relates the two programs is used to determine how this simplification should be done, as we explain in Section 4. If these simplified programs are themselves too complicated for symbolic analysis, they are simplified recursively using the same strategy until programs simple enough for symbolic analysis are obtained. This technique of recursive simplification and symbolic analysis is called fractal symbolic analysis. We have shown that fractal symbolic analysis is strictly more powerful than dependence analysis, provided the underlying symbolic analyzer satisfies some intuitively reasonable properties.

In Section 5, we demonstrate the effectiveness of fractal symbolic analysis in verifying the equivalence of left- and right-looking versions of LU with pivoting. Since fractal symbolic analysis is strictly more powerful than dependence analysis, it can also verify the legality of conversion between left-looking and right-looking forms of Cholesky factorization and triangular solve as well.

We conclude the paper with a discussion of future directions for this work.

2 Right-left Interchange

In this section, we show that right- and left-looking formulations of triangular solve, Cholesky factorization, and LU factorization with pivoting can be represented schematically as shown in Figure 5. We then discuss the transformation of the right-looking schema to the left-looking one, and vice versa. We will refer to this transformation as *right-left interchange*.

```

do k = 1,n
  B1(k);
  do j = k+1,n
    B2(k,j);

```

(a) Right-looking Code

```

do j = 1,n
  do k = 1,j-1
    B2(k,j);
  B1(j);

```

(b) Left-looking Code

Fig. 5. Right-left Interchange

2.1 Triangular Solve

The triangular solve versions shown in Figure 3 map directly to the template of Figure 5. Both B1 and B2 are represented by a single statement. B1 corresponds to the final scaling step of solving a single equation with one unknown, and B2 corresponds to the substitution of a solved unknown ($x(k)$) to compute an unsolved unknown ($x(j)$). Although triangular solve is often introduced in its left-looking form (as in [3]), the right-looking formulation in Fortran exhibits better spatial locality, and therefore performs better as shown in Figure 4(a).

2.2 Cholesky Factorization

The two formulations of Cholesky factorization shown in Figure 1 also map directly to the template of Figure 5. In this case, B1 and B2 are represented by small blocks of code. B1 corresponds to the computation in the current column, and B2 corresponds to updates from columns on the left to columns on the right. As in the case of triangular solve, performance is sensitive to the formulation that is used, as shown in Figure 4(b).

2.3 LU Factorization with Partial Pivoting

Converting between a right-looking formulation of LU with pivoting (Figure 6(a)) and a left-looking formulation (Figure 6(d)) is a more involved process than in the case of triangular solve or Cholesky factorization. Pivoting requires swapping the elements of two rows of the matrix, and can be viewed as a second ‘update’ operation. Conversion between right- and left-looking forms may be accomplished by two applications of right-left loop interchange. The update step in the right-looking code in Figure 6(a) may be converted to left-looking form as in Figure 6(b). Converting the swap to its left-looking form is slightly more complicated since the swap is never purely right-looking (pivoting requires swaps in earlier columns as well as latter columns). Nevertheless, the right-looking portion of the swap may be isolated by index-set splitting and statement reordering [9], as in Figure 6(c). A second application of right-left loop interchange then produces the left-looking LU factorization in Figure 6(d).

Figure 4(c) shows the performance of these codes on the SGI Octane. Although the right-looking version is simpler, the left-looking version has notably better performance.

```

do k = 1, N
  // Pick the pivot
  B1.a(k) :
    p(k) = k
  B1.b(k) :
    do i = k+1, N
      if abs(A(i,k)) > abs(A(p(k),k))
        p(k) = i

  // Swap rows
  B1.c(k) :
    do j = 1, N
      tmp = A(k,j)
      A(k,j) = A(p(k),j)
      A(p(k),j) = tmp

  // Scale current column
  B1.d(k) :
    do i = k+1, N
      A(i,k) = A(i,k) / A(k,k)

  // Update from current column
  // to columns to right
  do j = k+1, N
    B2(k,j) :
      do i = k+1, N
        A(i,j) = A(i,j) - A(i,k)*A(k,j)

```

(a) Right-looking LU

```

do j = 1, N
  // Apply delayed updates from left
  do k = 1, j-1
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows to the left
  do k = 1, j
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)

  // Swap rows to the right
  do k = j+1, N
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

```

(c) Hybrid Right-Left LU #2

```

do j = 1, N
  // Apply delayed updates from left
  do k = 1, j-1
    B2(k,j) :
      do i = k+1, N
        A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
  B1.a(j) :
    p(j) = j
  B1.b(j) :
    do i = j+1, N
      if abs(A(i,j)) > abs(A(p(j),j))
        p(j) = i

  // Swap rows
  B1.c(j) :
    do k = 1, N
      tmp = A(j,k)
      A(j,k) = A(p(j),k)
      A(p(j),k) = tmp

  // Scale current column
  B1.d(j) :
    do i = j+1, N
      A(i,j) = A(i,j) / A(j,j)

```

(b) Hybrid Right-Left LU #1

```

do j = 1, N
  // Apply delayed swaps from left
  do k = 1, j-1
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp

  // Apply delayed updates from left
  do k = 1, j-1
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows to the left
  do k = 1, j
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)

```

(d) Left-looking LU

Fig. 6. LU Factorization with Partial Pivoting

<pre>do k = 1,n B1(k); do j = k+1,n B2(k,j);</pre> <p>(a) Right-looking Code</p>	<pre>do k = 1,n do j = k,n if (j == k) B1(k); if (j != k) B2(k,j);</pre> <p>(b) After Code Sinking</p>
<pre>do j = 1,n do k = 1,j if (j == k) B1(k); if (j != k) B2(k,j);</pre> <p>(c) After Loop Interchange</p>	<pre>do j = 1,n do k = 1,j-1 B2(k,j); B1(j);</pre> <p>(d) After Loop Peeling: Left-looking Code</p>

Fig. 7. Converting Right-looking Code to Left-looking Code

2.4 Discussion of Right-left Interchange

The right-left interchange transformation in Figure 5 can be viewed as a combination of the more elementary transformations of code sinking, loop interchange, and loop peeling. To convert the right-looking version to the left-looking version, we sink statement B1 in Figure 7(a) into the inner loop to obtain the perfectly-nested loop shown in Figure 7(b). Performing perfectly-nested loop interchange produces the code shown in Figure 7(c). Finally, loop peeling produces the left-looking code shown in Figure 7(d). A similar sequence of elementary transformations converts left-looking code to right-looking code.

Code sinking and loop peeling are always legal, but loop interchange is illegal in some codes, so right-left interchange is not always legal. How does a compiler determine if this transformation is legal? We address this question next. In the rest of the paper, we will consider right-left interchange as a single transformation rather than as a composition of elementary transformations.

3 Dependence Analysis

In many programs, dependence analysis can be used to verify the legality of right-left interchange. In this section, we show that (i) dependence analysis is adequate to prove the legality of right-left interchange for triangular solve and Cholesky factorization, and (ii) it is not adequate to prove that this transformation is legal for LU factorization with pivoting.

3.1 Overview of Dependence Analysis

Two statements (more generally, statement instances) are said to be *dependent* if one of them may write to a memory location that may be read or written by the other. Statements (more generally, statement instances) that are not dependent are said to be *independent* [9]. A compiler that uses dependence analysis will assert that a transformation is legal if all pairs of statement instances that get reordered by the transformation can be shown to be independent.

Transformation	Legality Condition
Loop Interchange <pre> do i = 1,n do j = 1,m do j = 1,m <-> do i = 1,n B(i,j); B(i,j); </pre>	$independent((B(p, q), B(r, s)) :$ $1 \leq p < r \leq n \wedge 1 \leq s < q \leq m)$
Right-left interchange <pre> do k = 1,n do j = 1,n B1(k); do k = 1,j-1 do j = k+1,n <-> B2(k,j); B2(k,j); B1(j); </pre>	$independent((B1(t), B2(r, s)) :$ $1 \leq r < t < s \leq n)$ \wedge $independent((B2(p, q), B2(r, s)) :$ $1 \leq p < r < s < q \leq n)$

Fig. 8. Legality of Transformations: Dependence Analysis

Figure 8 shows these legality conditions for loop interchange, and for right-left interchange. It is easy to verify that statement instances $B(p, q)$ and $B(r, s)$ are reordered by loop interchange if and only if $p < r$ and $q > s$. Combining this with loop bounds information yields the legality condition for loop interchange shown in Figure 8. Similar considerations yield the legality test for right-left interchange.

3.2 Triangular Solve

In triangular solve, all dependences arise from reads and writes to vector x . Consider the flow-dependence from $B2(r, s)$ to $B1(t)$ in this code. This dependence is described by the following set of inequalities:

$$\begin{aligned}
 s &= t \text{ (same memory location in read and write)} \\
 r &< t \text{ (write before read)} \\
 1 &\leq t \leq n \text{ (loop bounds)} \\
 1 &\leq r \leq n \text{ (loop bounds)} \\
 r + 1 &\leq s \leq n \text{ (loop bounds)}
 \end{aligned}$$

From Figure 8, we see that this dependence would prevent right-left interchange if the following condition is also true:

$$1 \leq r < t < s \leq n$$

It is easy to see that the conjunction of inequalities does not have a solution, hence this dependence does not prevent right-left interchange.

In a similar manner, it can be shown that none of the other dependences in triangular solve prevent right-left interchange. Therefore, dependence analysis is adequate to verify the legality of right-left interchange for triangular solve.

3.3 Cholesky Factorization

The analysis for Cholesky factorization is similar to the analysis of triangular solve. Let us consider the flow-dependence that arises because $B2$ in Figure 1(a)

writes to $A(i, j)$ and reads from $A(j, k)$. This dependence can be described by the following set of inequalities, assuming that the write happens in iteration $(k = p, j = q, i = i_w)$, that the read happens in iteration $(k = r, j = s, i = i_r)$, and that \prec represents lexicographic order.

$$\begin{aligned}
i_w &= s \text{ (same location)} \\
q &= r \text{ (same location)} \\
1 &\leq p \leq N \text{ (loop bounds)} \\
p + 1 &\leq q \leq N \text{ (loop bounds)} \\
q &\leq i_w \leq N \text{ (loop bounds)} \\
1 &\leq r \leq N \text{ (loop bounds)} \\
r + 1 &\leq s \leq N \text{ (loop bounds)} \\
s &\leq i_r \leq N \text{ (loop bounds)} \\
(p, q, i_w) &\prec (r, s, i_r)
\end{aligned}$$

From Figure 8, we see that this dependence would prevent right-left interchange if the following condition is also true:

$$1 \leq p < r < s < q \leq n$$

It is trivial to verify that the conjunction of inequalities does not have a solution, so this dependence does not prevent right-left interchange.

In a similar manner, it can be shown that none of the other dependences prevent right-left interchange. We conclude that dependence analysis is adequate to verify the legality of right-left interchange for Cholesky factorization.

3.4 LU Factorization with Pivoting

We now show that dependence analysis is not adequate to verify the legality of right-left interchange for LU factorization with pivoting. The key problem is that in the right-looking version, swaps and updates to a given column are interleaved, whereas in the left-looking version, all delayed swaps to the column are performed before any updates are applied. Since both swaps and updates write to the column, dependences are violated by the right-left transformation.

In particular, consider the first application of right-left interchange that transforms the right-looking formulation of Figure 6(a) into the hybrid version shown in Figure 6(b). There is a flow-dependence from reference $A(i, j)$ in B2 to reference $A(k, j)$ in block B1.c which can be described by the following inequalities, assuming that the write takes place in iteration $(k = r, j = s, i = i_r)$ and the read takes place in iteration $(k = t, j = j_s)$.

$$\begin{aligned}
t &= i_r \text{ (same array location)} \\
j_s &= s \\
r &< t \text{ (lexicographic order)}
\end{aligned}$$

$$\begin{aligned}
& 1 \leq r \leq N \text{ (loop bounds)} \\
& r + 1 \leq s \leq N \\
& r + 1 \leq i_r \leq N \\
& 1 \leq t \leq N \\
& 1 \leq j_s \leq N
\end{aligned}$$

This dependence prevents right-left interchange if the following condition is also true:

$$1 \leq r < t < s \leq N$$

It is easy to verify that the conjunction of inequalities has solutions such as the following:

$$\begin{aligned}
r &= 2 \\
t &= i_r = 3 \\
s &= j_s = 4 \\
N &= 10
\end{aligned}$$

Therefore, a compiler that relies on dependence analysis will conclude that the first application of right-left interchange may not be legal. This conclusion holds even if the compiler analyzes the right-looking and left-looking formulations of Figures 6(a,d) directly; in the right-looking formulation, instance $(k = 2, j = 4, i = 3)$ of statement B2 is performed before instance $(k = 3, j = 4)$ of statement B1.c, whereas in the left-looking formulation, the order of these dependent instances is obviously reversed.

4 Fractal Symbolic Analysis

In this section, we give a brief overview of *fractal symbolic analysis*, a technique we proposed in [6] to establish legality of program transformations. As discussed earlier, dependence analysis is too conservative for codes such as LU factorization with pivoting, and symbolic analysis is generally impractical. Fractal symbolic analysis combines the tractability of dependence analysis with some of the power of symbolic analysis.

4.1 Overview of Fractal Symbolic Analysis

We assume that input programs consist of assignment statements and structured control flow (e.g., do-loops and conditionals). No unstructured control flow is allowed.

Figure 9 gives a high-level view of fractal symbolic analysis. Suppose S and T are an input program and its restructured version respectively. If S and T are “simple enough” to be compared symbolically, the symbolic analysis engine computes expressions for the outputs of these programs in terms of their inputs,

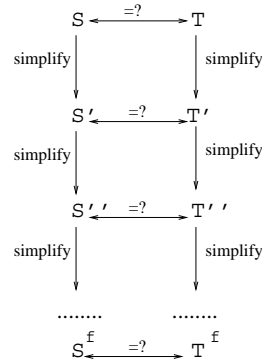


Fig. 9. Fractal Symbolic Analysis

and attempts to prove equality of these expressions. On the other hand, if the programs are not simple enough, fractal symbolic analysis generates two simplified programs S' and T' such that equality of these simplified programs implies equality of the original programs. The restructuring transformation that relates S and T is used to determine how the simplification must be carried out. If these simplified programs are still too complex to be analyzed symbolically, they themselves are simplified recursively until programs S^f and T^f that are simple enough to be analyzed symbolically are generated. The recursive simplification process that underlies our symbolic analysis technique is the motivation for calling it *fractal* symbolic analysis.

It can be shown that the recursive simplification process is guaranteed to produce programs that are simple enough to be analyzed by a symbolic analyzer that can analyze straight-line code [6]. However, there is a caveat. In general, equality of the simplified programs is sufficient but not necessary to guarantee equality of the original programs. Intuitively, each step of simplification produces programs that are less likely to be equal even if the original programs being compared are equal, so it is important to apply the simplification process sparingly. This means that the base symbolic analyzer must be as powerful as possible.

In the rest of this section, we describe (i) the simplification process and (ii) the base symbolic analysis engine used to compare sufficiently simplified programs.

4.2 Simplification

The simplification process we present is independent of the underlying symbolic analysis engine in the sense that the power of the symbolic analyzer only determines whether or not a program is simplified; if the program must be simplified, the rule for generating the simplified program is independent of the symbolic analyzer. The simplification process requires only that any symbolic analysis engine e has a corresponding predicate $Simple_e(p)$ associated with it such that $Simple_e(p)$ is true iff p can be analyzed by symbolic engine e . To keep notation

<pre> do i = 1,N do j = 1,N S(i,j) : z = z + A(i,j) (a) Reduction Loop </pre>	<pre> do j = 1,N do i = 1,N S(i,j) : z = z + A(i,j) (b) After Loop Interchange </pre>
---	---

Fig. 10. Loop Interchange of a Reduction Loop

<pre> S(i,j) : z = z + A(i,j) S(i',j') : z = z + A(i',j') (a) First ordering </pre>	<pre> S(i',j') : z = z + A(i',j') S(i,j) : z = z + A(i,j) (b) Second ordering </pre>
---	--

Fig. 11. Simplified Programs

simple, we will usually drop the subscript e if it is clear from the context. The technical results in this section depend only on three assumptions about such a predicate.

Definition 1. *A symbolic analysis engine is said to be adequate if the following statements are true.*

1. *A program consisting of a single assignment statement is Simple.*
2. *If a program pgm is not Simple, any program that contains pgm as a sub-program is itself not Simple.*
3. *If pgm_1 and pgm_2 are Simple, and there are no dependences between pgm_1 and pgm_2 , then $\{\text{pgm}_1; \text{pgm}_2\}$ and $\{\text{pgm}_2; \text{pgm}_1\}$ are Simple.*

The first two conditions are intuitively reasonable. Any symbolic analyzer should at least be able to analyze a program consisting of a single assignment statement. Moreover, if a program cannot be analyzed symbolically, a more complex program that contains that program within it is unlikely to be amenable to symbolic analysis.

The third condition is somewhat more subtle. Some symbolic analysis engines may be able to analyze programs pgm_1 and pgm_2 , but complex patterns of dependences between the two programs may prevent the symbolic analyzer from symbolically analyzing compositions of these programs such as $\{\text{pgm}_1; \text{pgm}_2\}$. However, if there are no dependences between these two programs, this interference does not arise, and it is reasonable to require that the symbolic analyzer be able to analyze $\{\text{pgm}_1; \text{pgm}_2\}$ and $\{\text{pgm}_2; \text{pgm}_1\}$, and prove that they are equal.

It can be shown that these three reasonable conditions make fractal symbolic analysis more powerful than dependence analysis. Note that even a symbolic analyzer that can only analyze straight-line code is adequate according to Definition 1, and therefore provides a more powerful analysis tool than dependence analysis.

As an example, consider the loop nest of Figure 10(a), and the loop nest shown in Figure 10(b) obtained by loop interchange. If addition is assumed to be commutative and associative, these two programs are equal. Note that

a compiler that uses dependence analysis will declare conservatively that the two programs are not equal; every instance of statement S reads and writes to location z , so the independence criterion in Figure 8 will not be satisfied (some compilers use pattern-matching to recognize reductions, but pattern-matching is notoriously fragile).

Suppose we have a simple symbolic engine capable of analyzing sequences of assignment statements. Our implementation of fractal symbolic analysis will proceed as follows. Since there are loops in the two programs, these programs are too complex to be analyzed directly by the given symbolic analyzer, so it is necessary to simplify them. The simplification rule for programs related by a loop interchange is shown in Figure 12. This rule is based on the intuitively reasonable idea that a "large-scale" transformation like loop interchange, which reorders a number of statement instances, can be viewed instead as a sequence of smaller transformations each of which reorders only one pair of statement instances. If each of the small transformations in such a sequence is legal, the "large-scale" transformation is obviously legal. It can be shown that such a sequence can be derived from the set of all pairs of statement instances that are reordered by the large-scale transformation [6]. Therefore, if the statement instances in every such pair can be executed in either order (they *commute*), the large-scale transformation is legal.

In Figure 11, legality of loop interchange can be established by demonstrating that any two instances $S(i, j)$ and $S(i', j')$ that are reordered by the transformation can be executed in either order without changing what is computed. This essentially requires us to prove that the two programs in Figure 11 are equivalent, subject to the conditions that $1 \leq i < i' \leq N$ and $1 \leq j' < j \leq N$. Note that these programs are simpler than the original programs; in particular, both of them are just sequences of assignment statements, and they can be analyzed by our symbolic analyzer.

The symbolic engine will attempt to prove them equivalent by demonstrating that the symbolic value of the output variable z , which we denote by z_{out} , is equal in both programs. In the first program, $z_{out} = ((z_{in} + A(i, j)) + A(i', j'))$ and in the second program, $z_{out} = ((z_{in} + A(i', j')) + A(i, j))$. If addition is commutative and associative, these two values are equal; therefore, the programs in Figure 11 are equal, so we can conclude that the programs in Figure 10 are themselves equal.

In this example, one application of the simplification rules was adequate to generate programs simple enough to be analyzed symbolically. More generally, the generated programs will themselves have to be simplified recursively.

Figure 12 shows the legality conditions for loop interchange and right-left interchange. It is useful to understand the similarities and the differences between the rules of Figure 8 and Figure 12. Independent program fragments obviously commute, but two program fragments may commute even if they are not independent. Intuitively, this is what makes fractal symbolic analysis more powerful than dependence analysis provided the symbolic analyzer is adequate. Application of the rules in Figure 12 produces two new programs which can be simplified

Transformation	Legality Condition
Loop Interchange $\begin{array}{l} \text{do } i = 1, n \\ \quad \text{do } j = 1, m \\ \quad \quad S(i, j); \end{array} \quad \leftrightarrow \quad \begin{array}{l} \text{do } j = 1, m \\ \quad \text{do } i = 1, n \\ \quad \quad S(i, j); \end{array}$	$\text{commute}(\langle S(p, q), S(r, s) \rangle : 1 \leq p < r \leq n \wedge 1 \leq s < q \leq m)$
Right/Left-looking Interchange $\begin{array}{l} \text{do } k = 1, n \\ \quad S1(k); \\ \quad \text{do } j = k+1, n \\ \quad \quad S2(k, j); \end{array} \quad \leftrightarrow \quad \begin{array}{l} \text{do } j = 1, n \\ \quad \text{do } k = 1, j-1 \\ \quad \quad S2(k, j); \\ \quad S1(j); \end{array}$	$\begin{array}{l} \text{commute}(\langle S1(t), S2(r, s) \rangle : \\ 1 \leq r < t < s \leq n) \wedge \\ \text{commute}(\langle S2(p, q), S2(r, s) \rangle : \\ 1 \leq p < r < s < q \leq n) \end{array}$

Fig. 12. Legality Conditions for Program Transformations

Commute Condition	Recursive Condition
Statement Sequence $\text{commute}(\langle s_1; s_2; \dots; s_N, B2 \rangle : \text{cond})$	$\begin{array}{l} \text{commute}(\langle S1, B2 \rangle : \text{cond}) \wedge \\ \text{commute}(\langle S2, B2 \rangle : \text{cond}) \wedge \\ \dots \\ \text{commute}(\langle S_N, B2 \rangle : \text{cond}) \end{array}$
Loop $\text{commute}(\langle \begin{array}{l} \text{do } i = 1, u \\ \quad S1(i); \end{array}, B2 \rangle : \text{cond})$	$\text{commute}(\langle S1(i), B2 \rangle : \text{cond} \wedge l \leq i \leq u)$
Conditional Statement $\text{commute}(\langle \begin{array}{l} \text{if } (\text{pred}) \text{ then} \\ \quad S1; \\ \text{else} \\ \quad S2; \end{array}, B2 \rangle : \text{cond})$	$\begin{array}{l} \text{commute}(\langle S1, B2 \rangle : \text{cond} \wedge \text{pred}) \wedge \\ \text{commute}(\langle S2, B2 \rangle : \text{cond} \wedge \neg \text{pred}) \end{array}$

Fig. 13. Recursive Simplification Rules

recursively using the rules in Figure 13. A detailed example of the application of the recursive simplification rules is provided by LU with pivoting which we discuss in the next section.

4.3 Symbolic Analysis and Comparison Engine

How powerful should the base symbolic analyzer be? In principle, even a base analyzer that can only handle straight-line code will endow a fractal symbolic analyzer with more power than dependence analysis. However, each step of recursive simplification generates programs that in general are less likely to be true even if the original programs are equal, so it is desirable to make the base analyzer as powerful as possible. In our experiments, we have found that it is desirable if the base analyzer can analyze the following class of programs.

Definition 2. *A program is simple if it has the following properties.*

1. *Array indices and loop bounds are affine functions of enclosing loop variables and symbolic constants, and predicates are conjunctions and disjunctions of affine inequalities of enclosing loop variables and symbolic constants.*
2. *No loop nest has a loop-carried dependence.*

Under these conditions, the value of each live variable at the end of a program, as a function of values at the start of the program, can be described by

$$A(\mathbf{k}) = \begin{cases} guard_1(\mathbf{k}) \rightarrow expression_1(\mathbf{k}) \\ guard_2(\mathbf{k}) \rightarrow expression_2(\mathbf{k}) \\ \vdots \\ guard_n(\mathbf{k}) \rightarrow expression_n(\mathbf{k}) \end{cases}$$

Fig. 14. Guarded Symbolic Expression

a *guarded symbolic expression* (called a *gse* for short) shown schematically in Figure 14. Each guard describes a polyhedral region of array indices, and the corresponding expression describes the values of the array elements for those indices. The symbolic analysis engine generates gse's for each live variable in the two programs to be compared, and compares the two gse's for each variable for equality.

Two gse's for a variable are considered to be equal if (i) the domain defined by the union of the guards in each gse are the same, and (ii) whenever the region defined by a guard in one gse has a non-empty intersection with the region defined by a guard in the other gse, the corresponding expressions for the value of the variable can be proved to be equal. In our current implementation, we consider two expressions for the value of a variable to be equal only if they are syntactically equal. This is adequate for our purpose, but more power would be obtained by using a symbolic algebra tool like Maple to exploit algebraic properties of operators like addition and multiplication in proving equality of expressions.

With these restrictions, computation and comparison of guarded symbolic expressions is a straightforward process and is described in detail in [6].

It is easy to see that a symbolic analyzer that can analyze simple programs, as defined in Definition 2, is adequate according to Definition 1. Therefore, the fractal symbolic analyzer described in this section is more powerful than a dependence analyzer.

5 LU with Pivoting

We now show that the fractal symbolic analyzer described in Section 4 deduces correctly that the right-looking and left-looking versions of LU with pivoting are semantically equal.

Figure 15 shows the steps involved in using fractal symbolic analysis to prove equivalence of the programs in Figure 6(a) and Figure 6(b). In the first step, the rule for right-left interchange in Figure 12 is used to generate the two commute conditions shown in the second column of Figure 15. Dependence analysis is adequate to verify one of the conditions, so we do not discuss it any further. The other condition is more involved; the two simplified programs corresponding to this condition are shown in Figure 16. We must show that these programs are equal subject to the condition that $(t \leq p(t) \wedge r < t < s)$.

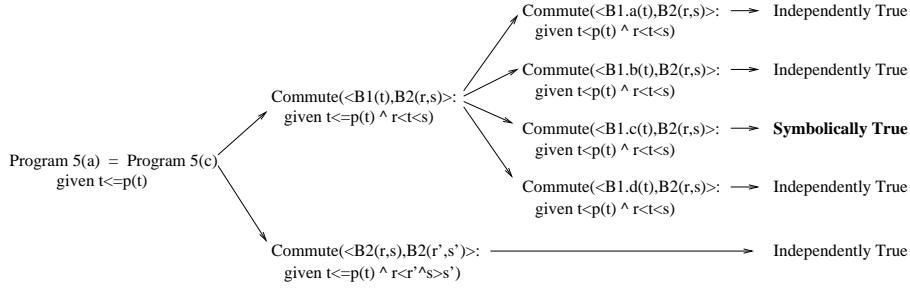


Fig. 15. Fractal Symbolic Analysis of LU

<pre> B2(r, s) : do i = r+1, N A(i, s) = A(i, s) - A(i, r)*A(r, s) B1.a(t) : p(t) = t B1.b(t) : do i = t+1, N if (abs(A(i, t) > abs(A(p(t), t)))) p(t) = i B1.c(t) : do k = 1, N tmp = A(t, k) A(t, k) = A(p(t), k) A(p(t), k) = tmp B1.d(t) : do i = t+1, N A(i, t) = A(i, t)/A(t, t) </pre> <p>(a) $B2(r, s); B1(t)$</p>	<pre> B1.a(t) : p(t) = t B1.b(t) : do i = t+1, N if (abs(A(i, t) > abs(A(p(t), t)))) p(t) = i B1.c(t) : do k = 1, N tmp = A(t, k) A(t, k) = A(p(t), k) A(p(t), k) = tmp B1.d(t) : do i = t+1, N A(i, t) = A(i, t)/A(t, t) B2(r, s) : do i = r+1, N A(i, s) = A(i, s) - A(i, r)*A(r, s) </pre> <p>(b) $B1(t); B2(r, s)$</p>
---	---

Fig. 16. After First Simplification Step

These programs are too complex to be analyzed symbolically because the loop that computes the pivot carries dependences. Therefore, they are simplified recursively using the first rule in Figure 13. This generates the four commute conditions shown in the third column of Figure 15. Dependence analysis is adequate to prove three out of four of these conditions.

The remaining commute condition, shown in Figure 17, involves the dependent update and swap operations, and cannot be verified by dependence analysis. At this point, however, the core symbolic analysis engine described in Section 4 can be used since none of the loops carry dependences. The only live, altered variable in either program is the array A , and the core symbolic engine generates *identical* guarded symbolic expressions for A from each program:

<pre> B2(r, s) : do i = r+1, N A(i, s) = A(i, s) - A(i, r)*A(r, s) B1.c(t) : do k = 1, N tmp = A(t, k) A(t, k) = A(p(t), k) A(p(t), k) = tmp </pre>	<pre> B1.c(t) : do k = 1, N tmp = A(t, k) A(t, k) = A(p(t), k) A(p(t), k) = tmp B2(r, s) : do i = r+1, N A(i, s) = A(i, s) - A(i, r)*A(r, s) </pre>
(a) $B2(r, s); B1.c(t)$	(b) $B1.c(t); B2(r, s)$

Fig. 17. After Second Simplification Step

$$A_{out}(i, j) = \begin{cases} i = l \wedge j = n & \rightarrow A_{in}(p(l), n) - A_{in}(p(l), m) * A_{in}(m, n) \\ i = p(l) \wedge j = n & \rightarrow A_{in}(l, n) - A_{in}(l, m) * A_{in}(m, n) \\ i = l \wedge j \neq n & \rightarrow A_{in}(p(l), j) \\ i = p(l) \wedge j \neq n & \rightarrow A_{in}(l, j) \\ i \neq l \wedge i \neq p(l) \wedge j = n & \rightarrow A_{in}(i, n) - A_{in}(i, m) * A_{in}(m, n) \\ i \neq l \wedge i \neq p(l) \wedge j \neq n & \rightarrow A_{in}(i, j) \end{cases}$$

In this example, the two gse's are syntactically identical. This demonstrates that the programs in Figure 17 (and, thus, the original codes in Figure 6(a) and 6(b)) are computationally equivalent. Because no assumptions were made about properties of arithmetic operators such as commutativity and associativity of addition and multiplication, we can conclude that these programs produce identical results even though machine arithmetic has finite precision.

This completes the verification that the programs of Figure 6(a) and Figure 6(b) are equal. Dependence analysis is adequate to verify that the program of Figure 6(b) is equal to the program of Figure 6(c) since the loop that scales the current column and the loop that completes the swap on rows to the right of the current column read and update disjoint locations. Finally, the transformation of the program of Figure 6(c) to the left-looking version of Figure 6(d) requires one application of right-left interchange, and its legality can be proved in a manner similar to that shown in Figure 15. We leave this final step to the interested reader.

5.1 Discussion

In Section 4, we argued that the fractal symbolic analyzer described there was strictly more powerful than a dependence analyzer. Therefore, the analyzer is adequate to verify the legality of the right-left transformation not only for LU factorization with pivoting but for triangular solve and Cholesky factorization as well.

6 Conclusions and Future Work

In this paper, we studied right-looking and left-looking formulations for three important linear algebra kernels, and argued that it is important to be able to restructure any formulation of these kernels into the other formulation. We showed that these restructurings were special cases of a general transformation that we called right-left interchange. We discussed how fractal symbolic analysis may be used to establish the legality of this transformation, and demonstrated its applicability to LU factorization with pivoting, a problem for which dependence analysis fails. Fractal symbolic analysis is the only technique general enough to prove equality of right-looking and left-looking formulations of all the examples in this paper.

There is a considerable body of related work on using symbolic analysis in restructuring compilers. Sophisticated symbolic analysis techniques for finding *generalized induction variables* have been developed by Haghghat and Polychronopoulos [4] and by Rauchwerger and Padua [7], but their concerns are very different from ours since they do not consider loop transformations. *Commutativity analysis* [8] is a program parallelization technique that uses symbolic analysis to determine if method invocations in object-oriented languages can be executed concurrently. This approach is based on the insight that a sequence of operations can be executed in parallel if each pair of operations can be performed in any order, provided adequate synchronization is introduced to protect access to shared variables. There is no analog of recursive simplification in commutativity analysis. Also, we are interested in proving the correctness of program transformations, not in parallelizing programs, and requiring all operations to commute with each other is too strong a condition for our application.

The focus of this paper has been on analysis. Synthesis of transformations is an interesting issue that we have not explored in the context of fractal symbolic analysis. Dependence information for loops can be represented abstractly using dependence vectors, cones, polyhedra etc., and these representations have been exploited to synthesize transformation sequences [2, 5, 1]. At present, we do not know suitable representations for the results of fractal symbolic analysis, nor do we know how to synthesize transformation sequences from such information. These issues remain to be investigated.

References

1. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000.
2. Uptal Banerjee. A theory of loop permutations. In *Languages and compilers for parallel computing*, pages 54–74, 1989.
3. Gene Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

4. Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.
5. Wei Li and Keshav Pingali. A singular loop transformation based on non-singular matrices. *International Journal of Parallel Programming*, 22(2), April 1994.
6. Nikolay Mateev, Vijay Menon, and Keshav Pingali. Fractal symbolic analysis. In *Proceedings of the 2001 International Conference on Supercomputing*, Sorrento, Italy, June 2001.
7. L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2), February 1999.
8. Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, November 1997.
9. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.