Left-looking to Right-looking and vice versa: An Application of Fractal Symbolic Analysis to Linear Algebra Code Restructuring

Nikolay Mateev, Vijay Menon, and Keshav Pingali

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

Abstract. We have recently developed a new program analysis strategy called *fractal symbolic analysis* that addresses some of limitations of techniques such as dependence analysis. 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: Cholesky factorization, LU factorization with pivoting, and triangular solve.

1 Introduction

Many computational science applications require the solution of linear systems of 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 algorithms of interest to us are (i) Cholesky factorization, (ii) LU factorization with partial pivoting, and (iii) triangular solve.

Substantial effort has been invested by the numerical analysis community in implementing high-performance versions of these algorithms. For example, the LAPACK library contains blocked implementations of these algorithms, optimized to perform well on a memory hierarchy [2]; the SCALAPACK library contains parallel implementations of these algorithms for distributed-memory machines [4].

In the compiler community, researchers have developed techniques to synthesize blocked and parallel implementations of these algorithms from high-level algorithmic formulations. These *restructuring* techniques perform source-to-source transformations to improve parallelism and locality of reference. A significant challenge for compiler optimization is the fact that there are many variations in how these algorithms can be expressed. The two most important variations are called *right-looking* or *eager*, and *left-looking* or *lazy*.

- *Right-looking*: In matrix factorization codes, the matrix is walked by column from left to right. After the current column has been computed, updates to columns to the right of the current column are performed immediately. Similarly in triangular solves, the current unknown is computed, and its contribution is immediately subtracted from the remaining equations. These are sometimes known as *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`.
- *Left-looking*: As in right-looking codes, the matrix is walked by column from left to right. At each step, updates to the current column from earlier elements/columns are performed, and the current column is then computed. Therefore, updates to a given column from earlier columns are performed as late as possible, which is why 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`.

Neither right- nor left-looking forms should be viewed as canonical. For example, Golub and van Loan's textbook on matrix computations [5] 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 2 demonstrates[1]. 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.

The most commonly used technique for proving legality of transformations is *dependence analysis* [12]. 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 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

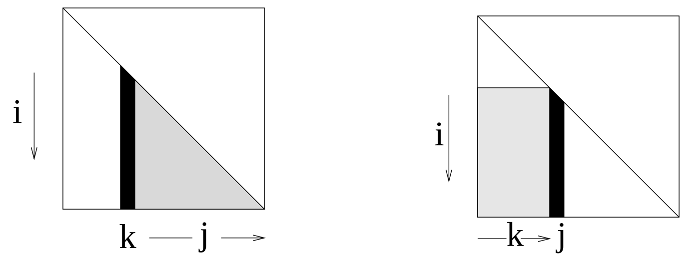
[1] These numbers were obtained on a 300 MHz SGI Octane with a 2MB L2 cache, and an R12K processor. All compiled code was generated using the SGI MIPSpro f77 compiler with flags: -O3 -n32 -mips4.

```
      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)
```



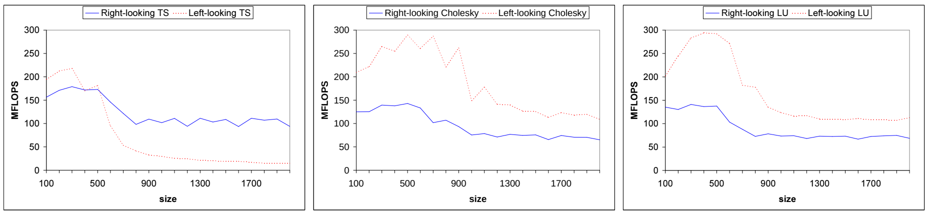
(a) Right-looking Cholesky (b) Left-looking Cholesky

Fig. 1. Cholesky Factorization

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. Unfortunately, a compiler that uses dependence analysis to validate transformations must conclude that left-looking and right-looking LU factorization codes are not semantically equivalent, as we discuss in Section 3. The essence of the problem is that dependence analysis provides a sufficient but not necessary condition for legality of a transformation.

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.

To bridge this gap between dependence analysis and symbolic analysis, we developed *fractal symbolic analysis* [9]. 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.



(a) Triangular Solve (b) Cholesky Factorization (c) LU with Pivoting

Fig. 2. Performance of Left- and Right-looking Codes

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*.

In this paper, we show how this new analysis technique can be used to prove the equality of left- and right-looking versions of triangular solve, Cholesky factorization, and LU factorization with partial pivoting. In Section 2, we discuss the transformations required to convert between left- and right-looking formulations of these algorithms. In Section 3, we show that dependence analysis is adequate to prove equality of left- and right-looking versions of triangular solve and Cholesky factorization, but that it is not adequate for LU factorization with pivoting. In Section 4, we describe fractal symbolic analysis briefly. In Section 5, we demonstrate its effectiveness in verifying the equivalence of left- and right-looking versions of LU with pivoting. We conclude with a discussion of future directions for our 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 3. 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*.

2.1 Lower Triangular Solve

Triangular solve, shown in Figure 4, maps directly to the template of Figure 3. 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

```
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. 3. Equivalent Right-looking and Left-looking Codes

```
          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)
```

(a) Right-looking Triangular Solve

```
          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)

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

(b) Left-looking Triangular Solve

Fig. 4. Lower Triangular Solve

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 [5]), the right-looking formulation in Fortran exhibits better spatial locality, and therefore performs better as shown in Figure 2(a).

2.2 Cholesky Factorization

Figure 1 presents the right- and left-looking versions of Cholesky factorization. The two formulations of Cholesky factorization map directly to the template of Figure 3. 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 before, performance is sensitive to the formulation that is used. However, the left-looking formulation results in better performance, as shown in Figure 2(b).

2.3 LU Factorization with Partial Pivoting

Our last example is LU factorization with partial pivoting. Converting between a right-looking formulation (Figure 5(a)) and a left-looking formulation (Figure 5(b)) 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. In the right-looking code in Figure 5(a), the update alone may be converted to left-looking form, as in Figure 5(c). Converting the swap 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

```
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
  // 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)

  // 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)
```

(b) Left-looking LU

```
do j = 1, N
  // Update to current column
  // from columns to 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)
```

(c) Hybrid Right-Left LU #1

```
do j = 1, N
  // Update to current column
  // from columns to 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
```

(d) Hybrid Right-Left LU #2

Fig. 5. LU Factorization with Partial Pivoting

```
do k = 1,n
  B1(k);
  do j = k+1,n
    B2(k,j);
```
(a) Right-looking Code

```
do k = 1,n
  do j = k,n
    if (j == k) B1(k);
    if (j != k) B2(k,j);
```
(b) After Code Sinking

```
do j = 1,n
  do k = 1,j
    if (j == k) B1(k);
    if (j != k) B2(k,j);
```
(c) After Loop Interchange

```
do j = 1,n
  do k = 1,j-1
     B2(k,j);
  B1(j);
```
(d) After Loop Peeling: Left-looking Code

Fig. 6. Converting Right-looking Code to Left-looking Code

and statement reordering [12], as in Figure 5(d). A second application of right-left loop interchange then produces the left-looking code in Figure 5(b).

Figure 2(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.

2.4 Right-left Interchange

The right-left interchange transformation in Figure 3 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 6(a) into the inner loop to obtain the perfectly-nested loop shown in Figure 6(b). Performing perfectly-nested loop interchange produces the code shown in Figure 6(c). Finally, loop peeling produces the left-looking code shown in Figure 6(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

We now discuss how dependence analysis can be used to verify legality of right-left interchange. We show that dependence analysis correctly deduces that right-left interchange is legal for triangular solve and Cholesky factorization, but it is not able to deduce 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

Transformation	Legality Condition

Loop Interchange
`do i = 1,n             do j = 1,m`
`  do j = 1,m      <->    do i = 1,n`
`    B(i,j);                B(i,j);`
	$independent(\langle B(p,q), B(r,s)\rangle :$
	$1 <= p < r <= n \;\wedge\; 1 <= s < q <= m)$

Right-left interchange
`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);`
	$independent(\langle B1(t), B2(r,s)\rangle :$
	$1 <= r < t < s <= n)$
	$\wedge$
	$independent(\langle B2(p,q), B2(r,s)\rangle :$
	$1 <= p < r < s < q <= n)$

Fig. 7. Legality of Transformations: Dependence Analysis

other. Statements (more generally, statement instances) that are not dependent are said to be *independent* [12].

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.

Figure 7 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 7. 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 7, we see that this dependence would prevent right-left interchange if the following condition is also true:

$1 <= r < t < s <= n$

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

In a similar manner, it can be shown that dependence analysis is adequate to verify legality of right-left interchange for Cholesky factorization. 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.

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

From Figure 7, we see that this dependence would prevent right-left interchange if the following condition is also true:
$1 <= p < r < s < q <= n$

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 5(a) into the hybrid version shown in Figure 5(c). 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 $(k = t, j = j_s)$.

$$
\begin{aligned}
t &= i_r \text{ (same array location)}\\
j_s &= s\\
r &< t \text{ (lexicographic order)}\\
1 <= &\ r <= N \text{ (loop bounds)}\\
r + 1 <= &\ s <= N\\
r + 1 <= &\ i_r <= N\\
1 <= &\ t <= N\\
1 <= &\ j_s <= N
\end{aligned}
$$

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

$1 <= r < t < s <= N$

It is easy to verify that the conjunction of inequalties 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 relied on dependence analysis would conclude that the first application of right-left interchange may not be legal. This conclusion holds even if the compiler analyzes the programs of Figure 5(a) and (b) 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 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 [9] 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 is combines the tractability of dependence analysis with some of the power of symbolic analysis.

4.1 Overview of Fractal Symbolic Analysis

Figure 8 shows how fractal symbolic analysis is used to analyze programs. Suppose `S` and `T` are a source program and its restructured version respectively. If `S`

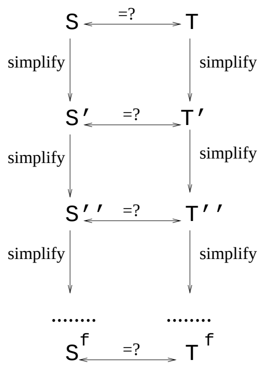


Fig. 8. Fractal Symbolic Analysis

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, 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 $\mathtt{S}^{\mathrm{f}}$ and $\mathtt{T}^{\mathrm{f}}$ 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. 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 the capabilities of our base symbolic analysis engine and the manner in which simplification is performed.

4.2 Symbolic Analysis and Comparison Engine

The symbolic analysis and comparison engine we use is relatively simple, but it is adequate for our purpose. It can symbolically analyze programs that satisfy the following conditions.

– Programs consist of assignment statements, for-loops and conditionals. The only data structures are scalars and dense arrays.

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

Fig. 9. Guarded Symbolic Expression

```
          do i = 1,N
            do j = 1,N
S(i,j) :      z = z + A(i,j)
```
(a) Reduction Loop

```
          do j = 1,N
            do i = 1,N
S(i,j) :      z = z + A(i,j)
```
(b) After Loop Interchange

Fig. 10. Loop Interchange of a Reduction Loop

- There are no loop-carried dependences.
- Array indices and loop bounds are restricted to be affine functions of enclosing loop variables and symbolic constants, and predicates are restricted to be conjunctions and disjunctions of affine inequalities.

Under these conditions, we have shown that 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 *guarded symbolic expression* (called a *gse* for short) shown schematically in Figure 9. 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 [9].

4.3 Simplification

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. 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 `k`, so the independence criterion in Figure 7 will not be satisfied (some compilers use pattern-matching to recognize reductions, but pattern-matching is notoriously fragile).

Our implementation of fractal symbolic analysis will proceed as follows. Since there are loop-carried dependences, it will simplify the two programs using the rule for loop interchange shown in Figure 11. Loop interchange reorders particular instances of statements. This reordering may be viewed *incrementally* by interchanging instances *one pair at a time.* In the example above, the legality of loop interchange is established by symbolically 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 (if so, we will say that these instances *commute*). This requires us to prove that $\mathtt{z}_{out}$ `= ((`$\mathtt{z}_{in}$ `+ A(i,j)) + A(i',j'))` and $\mathtt{z}_{out}$ `= ((`$\mathtt{z}_{in}$ `+ A(i',j')) + A(i,j))` are equivalent, which may be verified by a relatively simple symbolic engine that knows about the associativity of addition. Equality of the simplified programs implies equality of the original program, so the programs in Figures 10(a) and (b) will be declared to be equal.

This simple example highlights a number of points about fractal symbolic analysis. First, it should be noted that the application of the rule for interchange in Figure 11 generates simpler programs (the two statements for $\mathtt{z}_{out}$) from the more complex programs we started with. 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. Second, if the symbolic analyzer did not know about algebraic properties of addition, it would declare that the two expressions for $\mathtt{z}_{out}$ are not equal, and it would conclude conservatively that the programs of Figure 10(a),(b) are not equal. This partially illustrates the point that symbolic equality of the simplified programs is sufficient but in general not necessary for equality of the original programs. In this case, the simplified programs are, in fact, equivalent if commutativity of addition is allowed. More generally, the simplified programs need not be equivalent even if the original programs are.

Figure 11 shows the legality conditions for loop interchange and right-left interchange. It is useful to understand the similarity and the differences between the rules of Figure 7 and Figure 11. In particular, the application of the rules in Figure 11 produces two new programs which can simplified recursively using the rules in Figure 12, a process that has no analog in dependence analysis. A detailed example of the application of the recursive simplification rules is provided by LU with pivoting which we discuss next.

5 LU with Pivoting

We now demonstrate how fractal symbolic analysis is sufficient to verify the legality of right-left interchange for LU factorization with pivoting. In Section 2, we argued that two applications of right-left interchange are required to convert

Transformation	Legality Condition

Loop Interchange
`do i = 1,n`   `do j = 1,m`
`  do j = 1,m`   `<->`   `  do i = 1,n`
`    S(i,j);`   `    S(i,j);`
	$commute(\langle S(p,q), S(r,s)\rangle :$
	$1 <= p < r <= n \wedge 1 <= s < q <= m)$

Right/Left-looking Interchange
`do k = 1,n`   `do j = 1,n`
`  S1(k);`   `  do k = 1,j-1`
`  do j = k+1,n`   `<->`   `    S2(k,j);`
`    S2(k,j);`   `  S1(j);`
	$commute(\langle S1(t), S2(r,s)\rangle :$
	$1 <= r < t < s <= n)\wedge$
	$commute(\langle S2(p,q), S2(r,s)\rangle :$
	$1 <= p < r < s < q <= n)$

Fig. 11. Legality Conditions for Program Transformations

Commute Condition	Recursive Condition
Statement Sequence $commute(\langle$ `S1; S2;...; SN,B2` $\rangle : cond)$	$commute(\langle S1, B2\rangle : cond) \wedge$ $commute(\langle S2, B2\rangle : cond) \wedge$ $\ldots$ $commute(\langle SN, B2\rangle : cond)$
Loop $commute(\langle$ `do i = 1,u` `S1(i);` `,B2`$\rangle : cond)$	$commute(\langle S1(i), B2\rangle : cond \wedge l <= i <= u)$
Conditional Statement $commute(\langle$ `if (pred) then` `S1;` `else` `S2;` `,B2`$\rangle : cond)$	$commute(\langle S1, B2\rangle : cond \wedge pred) \wedge$ $commute(\langle S2, B2\rangle : cond \wedge \neg pred)$

Fig. 12. Recursive Simplification Rules

from a right-looking to a left-looking formulation. In Section 3, we saw that dependence analysis cannot verify the legality of the second application, i.e., it cannot prove that the programs in Figure 5(a) and 5(c) are equivalent. Dependence analysis fails because swaps (in `B1.c`) and updates (in `B2`) to the same column are reordered by right-left interchange. As a result, dependences from `B2` to `B1.c` are violated.

We now show that given the fact that $k \leq p(k)$, fractal symbolic analysis deduces correctly that the right-looking and left-looking versions of LU with pivoting are equal. The predicate $k \leq p(k)$ is easily inferred from the code using techniques such as array value propagation [8].

Figure 13 shows the steps involved in using fractal symbolic analysis to prove equivalence of the programs in Figure 5(a) and Figure 5(c). In the first step, the rule for right-left interchange in Figure 11 is used to generate the two commute conditions shown in the second column of Figure 13. Dependence analysis is adequate to verify the first condition, so we do not discuss it any further. The second condition is more involved; the two simplified programs corresponding to this condition are shown in Figure 14. We must show that these programs are equal subject to the condition that $(t \leq p(t) \wedge r < t < s)$.

These programs are too complex to be analyzed symbolically because the loop that computes the pivot carried dependences, so they are simplified recursively using the first rule in Figure 12. This generates the four commute conditions

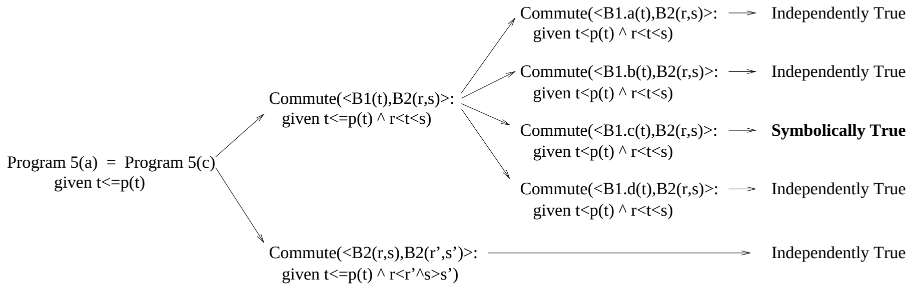


Fig. 13. Fractal Symbolic Analysis of LU

```
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)
```

(a) $B2(r,s); B1(t)$

```
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)
```

(b) $B1(t); B2(r,s)$

Fig. 14. After First Simplification Step

shown in the third column of Figure 13. Dependence analysis is adequate to prove three out of four of these conditions.

The remaining commute condition, shown in Figure 15, 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:

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

$B2(r,s)$:
```
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
```

(a) $B2(r,s); B1.c(t)$

$B1.c(t)$:
```
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)
```

(b) $B1.c(t); B2(r,s)$

Fig. 15. After Second Simplification Step

In this example, the two gse's are syntactically identical. This need not be the case in general. In this case, this demonstrates that the programs in Figure 15 (and, thus, the original codes in Figure 5(a) and 5(c)) are computationally equivalent. That is, fractal symbolic analysis is able to demonstrate that no floating point computation is reordered between right- and left-looking formulations of LU factorization with partial pivoting, so the transformation does not affect numerical stability.

This completes the verification that the programs of Figure 5(a) and Figure 5(c) are equal. Dependence analysis is adequate to verify that the program of Figure 5(c) is equal to the program of Figure 5(d) 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 5(d) to the left-looking version of Figure 5(b) requires one application of right-left interchange, and its legality can be proved in a manner similar to that shown in Figure 13. We leave this final step to the interested reader.

6 Conclusions

In this paper, we studied right and left formulations for three important linear algebra kernels, and argued the importance of automatically converting between the two formulations. Furthermore, we abstracted the high-level transformation that transforms one formulation to the other. 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 small body of related work on using symbolic analysis in restructuring compilers. Sophisticated symbolic analysis techniques for finding *generalized induction variables* have been developed by Haghighat and Polychronopoulos [6] and by Rauchwerger and Padua [10], but their concerns are very different from ours since they do not consider loop transformations. *Commutativity analysis* [11] 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. 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. There is also no analog of recursive simplification in commutativity analysis.

It should be clear that there are many design choices in implementing a fractal symbolic analyzer. In particular, we believe it would be useful to endow the core symbolic analyzer with knowledge about recurrences, and to let the symbolic algebra engine that proves equality of symbolic expression manipulate expressions using algebraic laws of addition and multiplication. We did not need this additional power for our purpose.

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 $[3, 7, 1]$. 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. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition*. SIAM, Philadelphia, 1995.
3. Uptal Banerjee. A theory of loop permutations. In *Languages and compilers for parallel computing*, pages 54–74, 1989.
4. L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - Design issues and performance. In *Supercomputing '96*. ACM SIGARCH and IEEE Computer Society, 1996.
5. Gene Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.
6. 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.
7. Wei Li and Keshav Pingali. A singular loop transformation based on non-singular matrices. *International Journal of Parallel Programming*, 22(2), April 1994.

8. V. Maslov. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *Proc. International Conference on Supercomputing*, pages 265–269, July 1995.
9. Nikolay Mateev, Vijay Menon, and Keshav Pingali. Fractal symbolic analysis for program transformations. Technical Report TR2000-1781, Cornell University, Computer Science, January 2000.
10. 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.
11. 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.
12. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.