Fractal Symbolic Analysis

Vijay Menon, *Intel Corporation*,
Keshav Pingali, *Cornell University*,
Nikolay Mateev, *Hewlett-Packard Corporation*.

Modern compilers perform wholesale restructuring of programs to improve their efficiency. Dependence analysis is the most widely used technique for proving the correctness of such transformations, but it suffers from the limitation that it considers only the memory locations read and written by a statement without considering what is being computed by that statement. Exploiting the semantics of program statements permits more transformations to be proved correct, and is critical for automatic restructuring of codes such as LU with partial pivoting.

One approach to exploiting the semantics of program statements is symbolic analysis and comparison of programs. In principle, this technique is very powerful, but in practice, it is intractable for all but the simplest programs.

In this paper, we propose a new form of symbolic analysis and comparison of programs which is appropriate for use in restructuring compilers. *Fractal* symbolic analysis is an *approximate* symbolic analysis that compares a program and its transformed version by repeatedly simplifying these programs until symbolic analysis becomes tractable while ensuring that equality of the simplified programs is sufficient to guarantee equality of the original programs.

Fractal symbolic analysis combines some of the power of symbolic analysis with the tractability of dependence analysis. We discuss a prototype implementation of fractal symbolic analysis, and show how it can be used to solve the long-open problem of verifying the correctness of transformations required to improve the cache performance of LU factorization with partial pivoting.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers and optimization*; I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Compilers, program optimization, program transformation, symbolic analysis

1. INTRODUCTION

Modern compilers perform source-level transformations of programs to enhance locality and parallelism. Before a program is transformed, it is analyzed to ensure that the transformation does not change the input-output behavior of that program. The most commonly used analysis technique is *dependence analysis* which computes the following partial order between execution instances of statements: a dependence is said to exist from a statement instance S_1 to a statement instance S_2 if (i) S_1 is executed before S_2, and (ii) one of these instances writes to a memory location that is read or written by the other one [Wolfe 1995]. Any reordering of statements consistent with this partial order is permitted since it

This work was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-0121401.
Corresponding author's address: K. Pingali, Department of Computer Science, Cornell University, Ithaca, NY 14853; email: pingali@cs.cornell.edu. This work was done while Vijay Menon and Nikolay Mateev were graduate students at Cornell University.
An earlier version of this paper was published in the 2001 International Conference on Supercomputing (ICS'01).

obviously leaves the input-output behavior of the program unchanged.

Dependence analysis has been the focus of much research in the past two decades. In early work, program transformation was carried out manually by the programmer, and dependence analysis was used only to verify the legality of the transformation [Cooper et al. 1986]. More recently, the research community has invented powerful algorithms for automatically synthesizing performance-enhancing transformations for a program from representations of its dependences, such as dependence matrices, cones, and polyhedra [Banerjee 1989; Feautrier 1992; Wolf and Lam 1991; Li and Pingali 1994]. These techniques are sufficiently well understood that they have been incorporated into production compilers such as the SGI MIPSPro [Wolf et al. 1996].

Dependence analysis provides *sufficient* but not *necessary* conditions for the legality of restructuring transformations. In his seminal 1966 paper, Bernstein pointed out that if f_1 and f_2 are *commutative* functions, the composition $f_1 \circ f_2$ can be restructured to $f_2 \circ f_1$ even though the restructuring violates dependences [Bernstein 1966]. The somewhat contrived example of Figure 1 provides a concrete illustration of this point. There is a dependence from statement `S1` to statement `S2` because `S1` writes to `y` and `S2` reads from `y`; similarly, there is a dependence from `S1` to `S3` because `S1` reads from `x` which is later written to by `S3`. There are only two statement reorderings consistent with this partial order: the original statement sequence, and the sequence obtained by permuting `S2` and `S3`. In particular, the statement sequence of Figure 1(b) is not consistent with this partial order. However, it is easy to see that if x_{in} and y_{in} are the values of `x` and `y` before execution of the statement sequences in Figure 1(a,b), the final values contained in variables `x` and `y` are $x_{in} * 2$ for both statement sequences; therefore, these two statement sequences are semantically equal. The statement reordering of Figure 1 is therefore legal, but a compiler that relies on dependence analysis alone will declare that this transformation is illegal. Note that the transformation in Figure 1 is legal even if the function symbol * is left uninterpreted. In practice, this means that the transformed program will produce the same output as the original program inspite of inaccuracies introduced by finite-precision arithmetic.

In conventional dependence analysis, a dependence is assumed to exist from a statement instance that writes to a variable `x` to a statement instance that reads from `x` even if there are intermediate statement instances that write to this variable. For some applications such as array privatization, it is necessary to identify the last write to a location that occured before that location is read by a particular statement instance. *Value-based* dependence analysis is a more precise variation of dependence analysis which computes this last-write-before-read information [Feautrier 1991]. It is easy to verify that the value-based dependence analysis computes the same partial order as standard dependence analysis for our example, so even a compiler that uses value-based dependence analysis will rule that the transformation is illegal.

Dependence analysis is limited because it considers only the sets of locations read and written by statements; in particular, because it does not consider what is being computed on the right-hand sides of statements. Notice that the dependences in Figure 1(a) do not change even if statement `S1` is changed to `y = x*x`, although statement reordering is not legal in the new program. Consideration of what is being computed by program statements can lead to a richer space of program transformations as is shown by our simple example, and is critical for restructuring important codes like LU with partial pivoting, as we discuss later in this paper.

Symbolic analysis [Gunter 1992] is the usual way of exploiting the semantics of program

```
    S1: y = x                   S3: x = x*2
    S2: y = y*2         =>      S2: y = y*2
    S3: x = x*2                 S1: y = x
(a) Original program        (b) Transformed program
```

Fig. 1. Simple Reordering of Statements

statements. To compare two programs for semantic equality, we derive symbolic expressions for the outputs of these programs as functions of their inputs, and attempt to prove that these expressions are equal. If an algebraic law such as the distributive, associative, or commutative law of addition and multiplication can be assumed by the compiler, operator symbols are interpreted appropriately and the corresponding laws may be used when proving expression equality. Symbolic analysis and comparison of programs is an extremely powerful technique for proving equality of programs; not only can it be used to verify the legality of program restructuring, but in principle, it can also be used to prove equality of programs that implement very different algorithms, such as sorting programs that implement quicksort and mergesort. In particular, the effect of the statement reordering shown in Figure 1 can be obtained by using techniques such as value numbering [Aho et al. 1986] which implicitly rely on symbolic evaluation. However, for all but the simplest programs, symbolic analysis and comparison is intractable.

In many areas of computer science, the barrier of intractability can be circumvented through the use of *approximation*. In this spirit, we have invented an approximate symbolic analysis which we call *fractal symbolic analysis*, and which is shown pictorially in Figure 2. We assume that we are given a symbolic analyzer that can symbolically analyze programs that are "simple enough". Depending on the power of the analyzer, these may be programs without loops, or programs with only DO-ALL loops, etc. Let `S` be a program that is to be restructured to a program `T`. If these programs are simple enough, we invoke the symbolic analyzer on these programs and either prove or disprove their equality. If these programs are too complex to be analyzed symbolically, we generate two simpler programs $\mathtt{S_1}$ and $\mathtt{T_1}$ which have a very important property: *equality of these programs is sufficient (although not necessary) for equality of the original programs*[1]. Analysis of the simpler programs is performed in a manner similar to that of the original programs: if the programs are simple enough, they are analyzed symbolically; otherwise, they in turn are simplified to produce new programs and so on (this is why we call our approach *fractal* symbolic analysis).

It is guaranteed that at some point, we will end up with programs $\mathtt{S_n}$ and $\mathtt{T_n}$ that are simple enough to be analyzed even by a symbolic analyzer that can only handle loop-free code. If we can prove that these programs are equal, we can conclude that `S` and `T` are equal; otherwise, we conservatively assume that `S` and `T` are not equal, and disallow the transformation.

A more abstract description of this method is the following. Let p be the predicate asserting semantic equality of the source and transformed programs. If we cannot prove p directly, we generate a simpler predicate q such that q implies p. If we can prove q, we conclude that p is true; if not, we assume conservatively that p is false and disallow the transformation. To prove the simpler predicate q, we apply this strategy recursively.

It should be clear from this description how the notion of approximation has been introduced into symbolic analysis.

[1] A small technical detail is that this step may produce a number of simpler programs from each of the original programs, in which case it is necessary to establish equality of that number of pairs of programs.

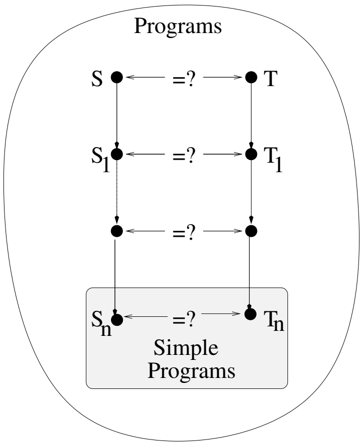


Fig. 2. Overview of Fractal Symbolic Analysis

Three important points should be noted.

(1) The rules for generating simpler programs are derived from the transformation that relates the two programs whose equality is to be established. Therefore, our approach cannot be used to prove that quicksort and mergesort are equal, for example, since these programs are not related by a restructuring transformation.

(2) Since equality of the simplified programs is a sufficient but not necessary condition for equality of the original programs, successive simplification steps produce programs that are less likely to be equal even if the original programs are equal. It is desirable therefore that the base symbolic analyzer be powerful so that recursive simplification can be applied sparingly.

(3) Fractal symbolic analysis is useful even if operators like + and * are left uninterpreted (that is, even if these operations cannot be assumed to obey the usual algebraic laws such as associativity and distributivity). For example, we use fractal symbolic analysis in this paper to show that the transformations required to block LU with pivoting are legal even if addition and multiplication do not obey algebraic laws.

The rest of this paper is organized as follows. In Section 2, we propose two challenge problems which cannot be solved using either dependence analysis or standard symbolic analysis. In Section 3, we introduce fractal symbolic analysis informally and show that it solves the two challenge problems. In Sections 4 and 5, we give a formal description of fractal symbolic analysis. Fractal symbolic analysis was invented for use in an ongoing project on automatic code restructuring for locality enhancement. In Section 6, we use fractal symbolic analysis to verify the correctness of transformations that are used to block LU with pivoting; we show that the resulting blocked code performs comparably to code in the LAPACK library [Anderson et al. 1995] on the SGI Octane. Finally, in Section 8, we discuss open problems and related work.

2. TWO CHALLENGE PROBLEMS

In this section, we discuss two challenge problems which we use to motivate fractal symbolic analysis. The first problem is concerned with restructuring reductions, and is rela-

```
do i = 1,M
  do j = 1,N
    B(i,j):
      k = k + A(i,j)
```

(a) Original Program

```
do j = 1,N
  do i = 1,M
    B(i,j):
      k = k + A(i,j)
```

(b) Transformed Program

Fig. 3. Loop Interchange of Reduction Code

tively simple. The second problem is distilled from LU with pivoting, and is much harder than the first.

2.1 First Challenge Problem

The program shown in Figure 3(a) adds the elements of an array `A` by traversing the array by rows. Loop permutation results in the version shown in Figure 3(b) in which the sum is computed by traversing the array by columns. We use the meta-variable `B(i,j)` to refer to instance `(i,j)` of the assignment statement in the loop body. If addition is assumed to be commutative and associative, these two loop nests are semantically equal, but if arrays are stored in column-major order, the version in Figure 3(b) will exhibit better spatial locality.

Each iteration of the loop nest reads and writes variable `k`, so there is a dependence from instance `B(i,j)` to all instances that follow it in execution order. Therefore, a compiler that relies on dependence analysis alone will rule that this transformation is illegal.

Restructuring reductions is important for obtaining good performance in scientific codes, so most restructuring compilers use some form of pattern-matching to recognize reductions, and to eliminate from consideration those dependences that arise from reductions. For example, the SGI MIPSPro compiler uses dependence analysis, but it is nevertheless capable of performing the loop interchange in Figure 3 because it appears to use some form of pattern-matching to identify reductions. However, pattern-matching is very fragile. For example, if temporaries are introduced into the program of Figure 3(a) as is shown in Figure 4, the SGI compiler fails to permute the loops even if `t1` and `t2` are not live after the loop nest.

```
do i = 1,M
  do j = 1,N
    B(i,j)://update on k
     t1 = k
     t2 = t1 + A(i,j)
     k = t2
```

Fig. 4. Introducing Temporaries into Program of Figure 3(a)

In principle, symbolic analysis and comparison can be used to prove the validity of the transformation shown in Figure 3, provided the analysis engine can reason about summations. Assuming that addition is associative, the output of the program of Figure 3(a) can be written as

$$k_{in} + \sum_{i=1}^{M} \sum_{j=1}^{N} A(i,j)$$

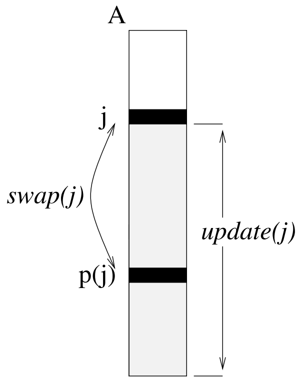
while the output of the program of Figure 7(b) can be written as

```
do j = 1,N-1
  B1(j): //swap
    tmp = A(j);
    A(j) = A(p(j));
    A(p(j)) = tmp;
  B2(j): //update
    do i = j+1,N
      A(i) = A(i)/A(j)
```

(a) Original Program

```
do j = 1,N-1
  B1(j): //swap
   tmp = A(j);
   A(j) = A(p(j));
   A(p(j)) = tmp;
do j = 1,N-1
  B2(j): //update
   do i = j+1,N
     A(i) = A(i)/A(j);
```

(b) Transformed Program



(c) Pictorial representation

Fig. 5. Second Challenge Problem

$$k_{in} + \sum_{j=1}^{N} \sum_{i=1}^{M} A(i, j).$$

If addition is commutative as well, the summations can be permuted, thereby proving the equivalence of the programs of Figure 3(a,b). However, as is the case with pattern-matching, the introduction of temporaries complicates symbolic analysis, and this approach may fail.

In Section 3, we show how fractal symbolic analysis solves this problem elegantly.

2.2 Second Challenge Problem

The second problem is a distillation of LU with partial pivoting, and is far more challenging than the first one. The source program of Figure 5(a) traverses an array `A`; at the $\mathtt{j}^{th}$ iteration, it swaps elements `A(j)` and `A(p(j))` (where `p(j)` $\geq$ `j`), and updates all the elements from `A(j+1)` through `A(N)` using the new value in `A(j)`. We assume that `p` is an array in which each entry `p(j)` is an integer between `j` and `N`.

This code is a much simplified version of LU factorization with partial pivoting in which entire rows of a matrix are swapped and entire sub-matrices are updated at each step. In our discussion, meta-variables `B1` and `B2` are used to refer to the swap and update statement blocks respectively. Figure 5(c) shows a pictorial representation of this program.

Loop distribution transforms this program into the one shown in Figure 5(b). In this program, all the swaps are done first, and then all the updates are done together. Are these programs equal?

Dependence analysis will assert that the transformation is legal if there is no dependence from an instance `B2(j2)` to an instance `B1(j1)` where `j1` $>$ `j2`. Unfortunately, this condition is violated: for any `j0` between `1` and `(N-2)`, instance `B2(j0)` writes to location `A(j0+1)`, and instance `B1(j0+1)` reads from it. Therefore, a compiler that relies on dependence analysis alone will disallow this transformation. Symbolic analysis of these programs on the other hand is too difficult.

In Section 3, we show how fractal symbolic analysis can be used to prove that the programs of Figure 5(a,b) are equal.

3. INFORMAL INTRODUCTION TO FRACTAL SYMBOLIC ANALYSIS

Although commutative subprograms do not seem to be an important class at this time, they will be treated briefly because of their relationship to the parallel situation. — A.J.Bernstein (1966).

In this section, we introduce informally the key ideas behind fractal symbolic analysis, using the two challenge problems to motivate the development.

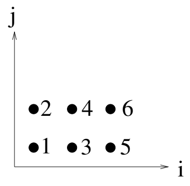
3.1 Incremental Transformation of Programs

A program transformation such as loop interchange or distribution is usually thought of as a metamorphosis that occurs in a single step. The insight behind fractal symbolic analysis is that it can be advantageous to view such a program transformation not as a single step but as the end result of a sequence of smaller transformations, each of which reorders only a small part of the entire computation. The advantage stems from the possibility that the smaller transformations may be easier to verify than the loop transformation itself. The idea therefore is similar to induction — if it is too difficult to prove a given predicate directly, induction may permit us to prove it "incrementally" by viewing the predicate as the end result of a sequence of simpler predicates which are easier to prove than the predicate itself.

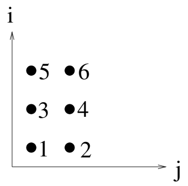
To understand this, let us consider the loop interchange shown in Figure 6(a,b). For convenience, instances of the loop body have been labeled with integers. In the original program, these instances are executed in the order [1 2 3 4 5 6], while in the transformed program, they are executed in the permuted order [1 3 5 2 4 6]. It is convenient to view such a permutation as a function p such that if $S = [S_1, S_2, ..., S_n]$ is the initial order of elements, the permuted order is $T = [S_{p(1)}, S_{p(2)}, ..., S_{p(n)}]$.

One way to decompose the transformation in Figure 6 into smaller transformations is to view it as the composition of a sequence of smaller transformations each of which reorders just two adjacent instances of the loop body (these are called *adjacent transpositions* in algebra [Johnson 1963]). Figure 6(c) shows two sequences of adjacent transpositions that convert the initial order of loop iterations to the final order. If every transposition along any *one* path from the initial order to the final order preserves program semantics, the loop interchange is obviously legal. In general, there are many sequences of adjacent transpositions that accomplish a given transformation, as the running example of Figure 6 shows, and our proof strategy may work for some of these sequences, and fail for others. This is similar to what happens in induction: some inductive proof strategies may succeed while others may fail.

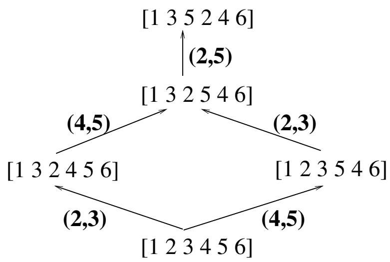
To find a sequence of adjacent transpositions that accomplishes a given permutation p, we exploit a standard result in combinatorial algebra. If S is a sequence of objects that is



(a) Source Program



(b) Transformed Program



(c) Loop Interchange by Adjacent Transpositions

Fig. 6. Incremental Loop Interchange

permuted to a sequence T by a permutation p, let $R(p)$ to be the set of pairs of elements in S that are reordered by that permutation; that is, $(S_i, S_j) \in R(p)$ if $(i < j) \wedge (p(i) > p(j))$. In our running example, R(p) = $\{$(2,3),(2,5),(4,5)$\}$. Figure 6(c) shows how these reordered pairs can be interpreted as adjacent transpositions to effect the loop interchange. More generally, we have the following result, many variations of which have appeared in the literature [Johnson 1963].

THEOREM 1. *Let $S = [S_1, S_2, ..., S_n]$ be a sequence of n objects and let $T = p(S)$ be a permutation of these objects where $T = [S_{p(1)}, S_{p(2)}, ..., S_{p(n)}]$. Define $R(p)$, the set of reordered pairs, as $R(p) = \{(S_i, S_j) : (1 \leq i < j \leq n) \wedge (p(i) > p(j))\}$.*

Sequence S can be transformed incrementally to sequence T by using the pairs in the set $R(p)$ as adjacent transpositions.

PROOF. The proof is an induction on the size of set *R(p)*.

- If $R(p) = \phi$, the result follows trivially.
- Assume inductively that for any sequence S' such that $T = p'(S')$ and $\|R(p')\| < m$, the inductive hypothesis is true. Consider a sequence S such that $T = p(S)$ and $\|R(p)\| = m$. There must be a pair $(S_k, S_{k+1}) \in R(p)$ such that $p(k) > p(k+1)$ (intuitively, this says that there must be at least one pair of adjacent elements in S that occur in the opposite order in T). Transposing S_k and S_{k+1} in S gives a sequence S' where $T = p'(S')$ for some permutation p', and $R(p') = R(p) - \{(S_k, S_{k+1})\}$. By inductive hypothesis, S' can be transformed incrementally to T using only adjacent transpositions from the set $R(p')$.

□

```
B(i1,j1):
  k = k + A(i1,j1)
B(i2,j2):
  k  = k + A(i2,j2)
```

(a){`B(i1,j1); B(i2,j2)`}

```
B(i2,j2):
  k =  k + A(i2,j2)
B(i1,j1):
  k =  k + A(i1,j1)
```

(b){`B(i2,j2); B(i1,j1)` }

Fig. 7. Loop Interchange of Reduction Code

The fact that a sequence of adjacent transpositions can accomplish any desired permutation is not surprising since it is well-known that the bubble-sort algorithm [Cormen et al. 1992] can sort any sequence by performing only adjacent transpositions. The significance of Theorem 1 is that it gives a closed-form characterization of such transpositions.

To verify the legality of a program transformation, we see that it is sufficient to show that for all pairs of statement instances (S_m, S_n) that are reordered by the transformation, the statement sequence $\{S_m; S_n\}$ is semantically equal to the statement sequence $\{S_n; S_m\}$ (that is, S_n and S_m *commute*). Intuitively, this guarantees that *all* sequences of adjacent transpositions which accomplish the program transformation will preserve program semantics. Although this is a stronger condition than we need, it is adequate for our purpose. For future reference, we state this result formally.

THEOREM 2. *Given a program G, let p be a program transformation, and let $R(G, p)$ be the set of statement instances in G that are reordered by p. Transformation p is legal if $\{S_m; S_n\}$ is equal to $\{S_n; S_m\}$ for all pairs $(S_m, S_n) \in R(G, p)$.*

PROOF. Follows immediately from Theorem 1. □

3.2 First Challenge Problem

Let us apply these ideas to verify the loop interchange in Figure 3. If the symbolic analyzer cannot analyze these programs, we generate simpler programs by considering the pairs of loop body instances that are reordered by the transformation. It is obvious that these are all pairs $\{\texttt{B(i1,j1),B(i2,j2):(1}\leq\texttt{i1}<\texttt{i2}\leq\texttt{M)}\wedge\texttt{(1}\leq\texttt{j2}<\texttt{j1}\leq\texttt{N)}\}$. Therefore, we need to show that the two statement sequences in Figure 7 are semantically equal.

Even a symbolic analyzer that is capable of analyzing only straight-line code can determine that the value of k after execution of the two statement sequences of Figure 7(a,b) is $((k_{in} + A(i1, j1)) + A(i2, j2))$, and $((k_{in} + A(i2, j2)) + A(i1, j1))$ respectively. If addition is assumed to be commutative and associative, these two expressions are equal (independent of the bounds on `i1, j1, i2` and `j2`), and the two programs of Figure 7 are semantically equal. Therefore, we deduce that the loop interchange in Figure 3 is legal. If addition is not commutative and associative, the two expressions are not equal, and we disallow the transformation.

Since fractal symbolic analysis is not based on pattern matching, more complex reductions such as the one in Figure 4 do not confuse the analysis. To verify that loop interchange is legal in Figure 4, the fractal symbolic analyzer generates the two programs shown in Figure 8. If *k* is the only variable live after the loop nest, the symbolic analyzer can easily deduce that both programs are equal, assuming that addition is commutative and associative.

```
B(i1,j1):
  t1 = k
  t2 = t1+ A(i1,j1)
  k = t2
B(i2,j2):
  t1 = k
  t2 = t1 + A(i2,j2)
  k = t2
```

(a) {`B(i1,j1); B(i2,j2)`}

```
B(i2,j2):
  t1 = k
  t2 = t1 + A(i2,j2)
  k = t2
B(i1,j1):
  t1 = k
  t2 = t1 + A(i1,j1)
  k = t2
```

(b) {`B(i2,j2); B(i1,j1)`}

Fig. 8. Verifying Loop Interchange

```
B1(j1): //swap
  tmp = A(j1);
  A(j1) = A(p(j1))
  A(p(j1)) = tmp;
B2(j2): //update
  do i = j2+1,N
    A(i) = A(i)/A(j2)
```

(a) {`B1(j1);B2(j2)`}

```
B2(j2)://update
  do i = j2+1,N
    A(i) = A(i)/A(j2)
B1(j1)://swap
  tmp = A(j1)
  A(j1) = A(p(j1))
  A(p(j1)) = tmp
```

(b) {`B2(j2);B1(j1)`}

Fig. 9. Simplified Programs for Second Challenge Problem

3.3 Second Challenge Problem

Fractal symbolic analysis of the second challenge program (Figure 5) reveals a few additional subtleties that are worth observing. The set of loop body instances that are reordered by the transformation is { `(B2(j2),B1(j1)):1`$\leq$`j2`$<$`j1`$\leq$`N`}. Therefore, the transformation is legal if the statement sequence {`B1(j1);B2(j2)`} (shown in Figure 9(a)) is semantically equal to the statement sequence {`B2(j2);B1(j1)`} (shown in Figure 9(b)) for `1`$\leq$`j2`$<$`j1`$\leq$`N`. Notice that (i) these programs are simpler than the ones in Figures 5(a) and (b) since each one has one less loop, and (ii) their equality implies equality of the original programs, as required by Figure 2.

The symbolic analyzer we use in our implementation can analyze programs with straight-line code and DO-ALL loops, so it can perform analysis and comparison of the programs in Figure 9(a,b) without any further simplification. Let A_{in} and A_{out} be the values in array `A` before and after the execution of the program in Figure 5(a). It is easy to see that A_{out} can be expressed in terms of A_{in} as a *guarded symbolic expression* (GSE for short), shown in Figure 10, consisting of a sequence of guards defining array regions and symbolic expressions for the values in the array in those regions. Each guard consists of conjunctions and disjunctions of affine function of loop variables and constants. Note that the array region defined by such a guard is not necessarily convex (for example, the last guard can be written as $(j2 < k \leq N) \wedge (k \neq j1) \wedge (k \neq p(j1))$ and the corresponding array region is obviously not convex).

$$A_{out}(k) = \begin{cases} 1 \leq k \leq j2 & \rightarrow \quad A_{in}(k) \\ k = j1 & \rightarrow \quad A_{in}(p(j1))/A_{in}(j2) \\ k = p(j1) & \rightarrow \quad A_{in}(j1)/A_{in}(j2) \\ else & \rightarrow \quad A_{in}(k)/A_{in}(j2) \end{cases}$$

Fig. 10. Guarded Symbolic Expression for A_{out}

```
B1(j1)://swap
  tmp = A(j1);
  A(j1) = A(p(j1));
  A(p(j1)) = tmp;
B2(j2,i)://update body
  A(i) = A(i)/A(j2);
```

(a){`B1(j1);B2(j2,i)`}

```
B2(j2,i)://update body
  A(i) =   A(i)/A(j2);
B1(j1)://swap
  temp = A(j1);
  A(j1) = A(p(j1));
  A(p(j1)) = tmp;
```

(b){`B2(j2,i);B1(j1)`}

Fig. 11. Another Step of Simplification

An identical GSE expresses the result of executing the program of Figure 5(b). If A is assumed to be the only live variable after execution of the two programs, we conclude that the programs of Figure 9(a) and (b) are equal, so the programs of Figure 5(a) and (b) are also equal. Moreover, since the two GSE's are syntactically equal, the programs of Figure 5(a) and (b) are equal even if the division operator is left uninterpreted.

3.4 Discussion

It is useful to understand what happens if we apply another step of simplification to the programs of Figure 9(a,b). The reordering of block `B1(j1)` over the iterations of loop `B2(j2)` can be viewed incrementally as a process in which block `B1(j1)` is moved over the iterations of loop `B2(j2)` one iteration at a time. If each move is legal, the entire reordering is clearly legal. The simplified programs are shown in Figure 11(a) and (b); we must verify that {`B1(j1);B2(j2,i)`} is equal to {`B2(j2,i);B1(j1)`}, assuming that N>j1>j2≥1 and that N≥i>j2.

However, it is easy to verify that these programs are not equal. For example, for $k = i = j1$, the final values in $A_{out}(k)$ are different. This illustrates the point discussed in Section 1. Equality of the simplified programs is a sufficient but not in general necessary condition for equality of the original programs, so the simplification process that is at the heart of fractal symbolic analysis should be applied sparingly. In particular, had our base symbolic analyzer been able to analyze only straight-line code, we would have concluded conservatively that the loop distribution of Figure 5(a,b) is not legal.

To formalize the intuitive ideas presented in this section, we need to answer two questions.

(1) How do we simplify programs?

(2) What base symbolic analyzer is both powerful and practical?

We answer these questions next.

4. SIMPLIFICATION RULES

In this section, we describe how programs are simplified when they are too complex to be analyzed symbolically.

We assume that input programs consist of assignment statements, do-loops and conditionals. No unstructured control flow is allowed. It is straight-forward to add other control-flow constructs such as case statements, but we will not do so to keep the discussion simple. The only data structures are multi-dimensional, flat arrays whose elements can be modified using assignment statements.

The simplification rules we present in this section are 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. To emphasize this point, we will describe the simplification process abstractly, assuming only that each 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 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* $\{pgm_1; pgm_2\}$ *and* $\{pgm_2; pgm_1\}$ *are* Simple *and provably equal.*

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 $\{pgm_1; pgm_2\}$. 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 $\{pgm_1; pgm_2\}$ and $\{pgm_2; pgm_1\}$, and prove that they are equal.

We will see later in this section that these three reasonable conditions make fractal symbolic analysis more powerful than dependence analysis.

4.1 Simplification Rules for Common Transformations

Table I(a) shows the rules used by the compiler to determine the legality of a few common transformations. To verify legality of a transformation shown in the first column, the *Commute* function is invoked on two simplified programs, as shown in the second column of this table. Constraints known to be satisfied by free variables of the simplified programs are also passed to this function. If the two simplified programs can be proved to commute given these constraints, the function returns true, and the transformation is legal; otherwise, the function returns false, and the compiler assumes conservatively that the transformation is not legal.

Of the transformations listed in this table, only *imperfectly-nested loop interchange* is somewhat non-standard. It is useful for converting so-called *right-looking* numerical codes such triangular solve and LU with pivoting to the corresponding *left-looking* versions, and vice versa [Mateev et al. 2000]. The effect of this transformation can be obtained by a combination of code-sinking, perfectly-nested loop interchange, and loop peeling, as shown in Figure 12. Only perfectly-nested loop interchange reorders computations, and it is easy to see that it reorders precisely the computations shown in the second column in Table I. Considering imperfectly-nested loop interchange to be a single transformation is therefore only a matter of convenience.

Transformation	Legality Condition
Statement Reordering S1; S2; <-> S2; S1;	$Commute(\langle S1, S2\rangle)$

Loop Distribution/Jamming
do i = 1,n
  S1(i);
  S2(i);
<->
do i = 1,n
  S1(i);
do i = 1,n
  S2(i);
	$Commute(\langle S1(i_1), S2(i_2)\rangle : 1 <= i_2 < i_1 <= n)$

Loop Interchange
do i = 1,n
  do j = 1,m
    S(i,j);
<->
do j = 1,m
  do i = 1,n
    S(i,j);
	$Commute(\langle S(i_1, j_1), S(i_2, j_2)\rangle :$
	$1 <= i_1 < i_2 <= n \wedge 1 <= j_2 < j_1 <= m)$

Linear Loop Transformation T do (i1,i2,...,ik) S(i1,i2,...,ik); <-> do (i1',i2',...,ik') S'(i1',i2',...,ik') where (i1',i2',...,ik') = T(i1,i2,...,ik)	$Commute(\langle S(\vec{i}), S(\vec{j})\rangle :$ $\vec{i} \prec \vec{j} \wedge \mathtt{T}(\vec{i}) \succ \mathtt{T}(\vec{j}))$[a] [a] The $\prec$ and $\succ$ denote the lexicographic ordering relation.

Imperfectly-nested Loop Interchange
do k = 1,n
  S1(k);
  do j = k+1,n
    S2(k,j);
<->
do j = 1,n
  do k = 1,j-1
    S2(k,j);
  S1(j);
	$Commute(\langle S1(t), S2(r, s)\rangle :$
	$1 <= r < t < s <= n) \wedge$
	$Commute(\langle S2(p, q), S2(r, s)\rangle :$
	$1 <= p < r < s < q <= n)$

(a) Simplification Rules for Common Loop Transformations

Commute Condition	Recursive Condition
Statement Sequence $Commute(\langle$ S1; S2;...; SN,pgm2 $\rangle : cond)$	$Commute(\langle S1, pgm2\rangle : cond) \wedge$ $Commute(\langle S2, pgm2\rangle : cond) \wedge$ ... $Commute(\langle SN, pgm2\rangle : cond)$
Loop $Commute(\langle$ do i = l,u S1(i); ,pgm2 $\rangle : cond)$	$Commute(\langle tmp1 = l, pgm2\rangle : cond) \wedge$ $Commute(\langle tmp2 = u, pgm2\rangle : cond) \wedge$ $Commute(\langle S1(i), pgm2\rangle : cond \wedge l <= i <= u)$

Conditional Statement
$Commute(\langle$ if (pred) then
  S1;
else
  S2; ,pgm2 $\rangle : cond)$
	$Commute(\langle tmp = pred, pgm2\rangle : cond) \wedge$
	$Commute(\langle S1, pgm2\rangle : cond \wedge pred) \wedge$
	$Commute(\langle S2, pgm2\rangle : cond \wedge \neg pred)$

(b) Recursive Simplification Rules

Table I. Simplification Rules

The intuition behind the legality conditions in Table I(a) has been described in Section 3 — the transformation is viewed as a composition of adjacent transpositions derived from the set of pairs of statement instances reordered by that transformation, so proving legality of the transformation is reduced to proving legality of each of the adjacent transpositions. Legality conditions for transformations not shown in Table I can be derived easily from this general principle.

```
do k = 1,n
  S1(k);
  do j = k+1,n
    S2(k,j);
```

(a) Source Program

```
do k = 1,n
  do j = k,n
    if (j==k) S1(k);
    if (j!=k) S2(k,j);
```

(b) After code-sinking: perfectly-nested loop

```
do j = 1,n
  do k = 1,j
    if (j==k) S1(k);
    if (j!=k) S2(k,j);
```

(c) After perfectly-nested loop interchange

```
do j = 1,n
  do k = 1,j-1
    S2(k,j);
  S1(j);
```

(d) After loop-peeling

Fig. 12. Decomposition of Imperfectly-nested Loop Interchange

4.2 Recursive Simplification

Given two programs pgm_1 and pgm_2 as input, the *Commute* function returns true if it can show that the program $\{pgm_1; pgm_2\}$ is equal to the program $\{pgm_2; pgm_1\}$ given the constraints specified on the free variables of pgm_1 and pgm_2. It first checks to see if $\{pgm_1; pgm_2\}$ and $\{pgm_2; pgm_1\}$ are *Simple*. If so, it invokes the symbolic analysis and comparison engine directly. Otherwise, it tries to generate simpler programs by simplifying pgm_1 and pgm_2 recursively.

Recursive simplification is performed in a syntax-driven fashion, using the rules in Table I(b). To avoid clutter in Table I(b), we have omitted the symmetric simplification rules for simplifying pgm_2 when it is not simple enough. The intuition behind the rules of Table I(b) has been described in Section 3 — we view the transformation of program $\{pgm_1; pgm_2\}$ into program $\{pgm_2; pgm_1\}$ as a composition of adjacent transpositions. For example, program $\{S1; S2; ...; SN; pgm_2\}$ is transformed to program $\{pgm_2; S1; S2; ...; SN\}$ by moving pgm_2 over SN,...,$S1$ successively.

The first rule in Table I(b) requires elaboration. Suppose $pgm1$ is a sequence of `N` statements. At one extreme, in a "coarse-grain" application of the rule, $pgm1$ can be partitioned into exactly two statement sequences `S1` and `S2` where both `S1` and `S2` are themselves statement sequences. At the other extreme, in a "fine-grain" application of the rule, $pgm1$ may be broken into `S1; S2; ...; SN` where none of `S1`, `S2`, ..., or `SN` is a statement sequence. Coarse-grain application of the rule is ambiguous because there are usually many different ways of splitting a sequence of statements into two subsequences. Fine-grain application is unambiguous, but it results in a stricter test, as we discuss later in this section. Our implementation applies this rule in a fine-grain manner, and we will assume this in the rest of the paper. With this caveat, the rules in Table I(b) are unambiguous.

4.3 Properties of Recursive Simplification Rules

Before we write a program to implement the recursive simplification rules, we must address issues such as termination, and determinacy of rule application. For this, it is useful to view the rules in Table I(b) as *rewrite rules* in a *term rewriting system*(TRS) [J.W.Klop 1980]. In our context, terms are conjunctions of commute conditions, and the rewrite rules are the rules of Table I(b). Figure 13 shows examples of terms and rewriting of terms. We address the following questions in this section.

(1) Does recursive simplification always terminate? This is called *strong normalization* in the TRS literature [J.W.Klop 1980].

(2) Suppose we are testing commutativity of pgm_1 and pgm_2. If neither of these programs is *Simple*, we can choose to simplify either of them first; we can even interleave the simplifications of pgm_1 and pgm_2. Is the final conjunction of commute conditions independent of these choices? This is called the Church-Rosser property in the TRS literature [J.W.Klop 1980].

Both questions are answered affirmatively in this section. Readers willing to take these answers on faith can omit the rest of this sub-section without loss of continuity.

Strong normalization follows from the fact that each application of the rewrite rules produces syntactically simpler programs. Repeated application of the rewrite rules ultimately produces programs that are single assignment statements; by Definition 1(1), these are *Simple*, so rewriting always terminates. This assertion is obviously true even if *Simple* programs can contain larger grain structures such as some conditionals and loops; in that case, the rewriting will simply terminate earlier.

The proof of the Church-Rosser property is straight-forward if tedious, and relies on a well-known result called Newman's Lemma from the TRS literature [J.W.Klop 1980]. We state the formal results.

DEFINITION 2. *Let A* = $(C, \rightarrow)$ *be a TRS where* C *is the set of terms and* $\rightarrow$ *is the rewrite relation. Let* $\stackrel{*}{\rightarrow}$ *be the reflexive, transitive closure of* $\rightarrow$.
A is said to be Church-Rosser if for all $a, b, c \in C$,
$(a \stackrel{*}{\rightarrow} b) \wedge (a \stackrel{*}{\rightarrow} c) \Rightarrow \exists d \in C.(b \stackrel{*}{\rightarrow} d) \wedge (c \stackrel{*}{\rightarrow} d)$.
A is said to be weakly *Church-Rosser if for all* $a, b, c \in C$,
$(a \rightarrow b) \wedge (a \rightarrow c) \Rightarrow \exists d \in C.(b \stackrel{*}{\rightarrow} d) \wedge (c \stackrel{*}{\rightarrow} d)$.

LEMMA 1. *(Newman's Lemma): Let A* = $(C, \rightarrow)$ *be a TRS. If A is weakly Church-Rosser and has the strong normalization property, then A is Church-Rosser.*

To prove that the rules of Table I(b) are weakly Church-Rosser, we have to consider terms of the form $Commute(\langle pgm_1, pgm_2 \rangle\ :\ cond)$ where neither pgm_1 and pgm_2 is *Simple*, because these are the terms that can be rewritten in multiple ways. There are nine cases to consider since pgm_1 and pgm_2 can each be either a statement sequence, a do-loop, or a conditional. In each case, we have to show that if b is the conjunction of commute conditions that result from rewriting pgm_1 first, and c is the conjunction of commute conditions that result from rewriting pgm_2 first, we can always rewrite b and c to reach the same conjunction of commute conditions d. The proof is straight-forward; Figure 13 shows the proof for the case when pgm_1 is a statement sequence and pgm_2 is a do-loop. The only subtle point is that by assumption, the statement sequence $S1; S2; ..., SN$ and the do-loop $do\ i =\ l, u\ S(i)$ are not *Simple*; therefore, from Definition 1(2), a program such as $\{S1; do\ i\ = l, u\ S(i)\}$ is not *Simple*, so commutativity conditions such as $Commute(\langle S1, do\ i =\ l, u\ S(i)\rangle : cond)$ can legitimately be rewritten by using the rule for simplifying do-loops.

From Newman's lemma, it follows that the rewrite rules of Table I(b) are Church-Rosser.

4.4 Implementation of Recursive Simplification

In our compiler, the *Commute* function is implemented by a recursive function as shown in Figure 14. It is passed two programs pgm_1 and pgm_2, some optional *bindings* which are

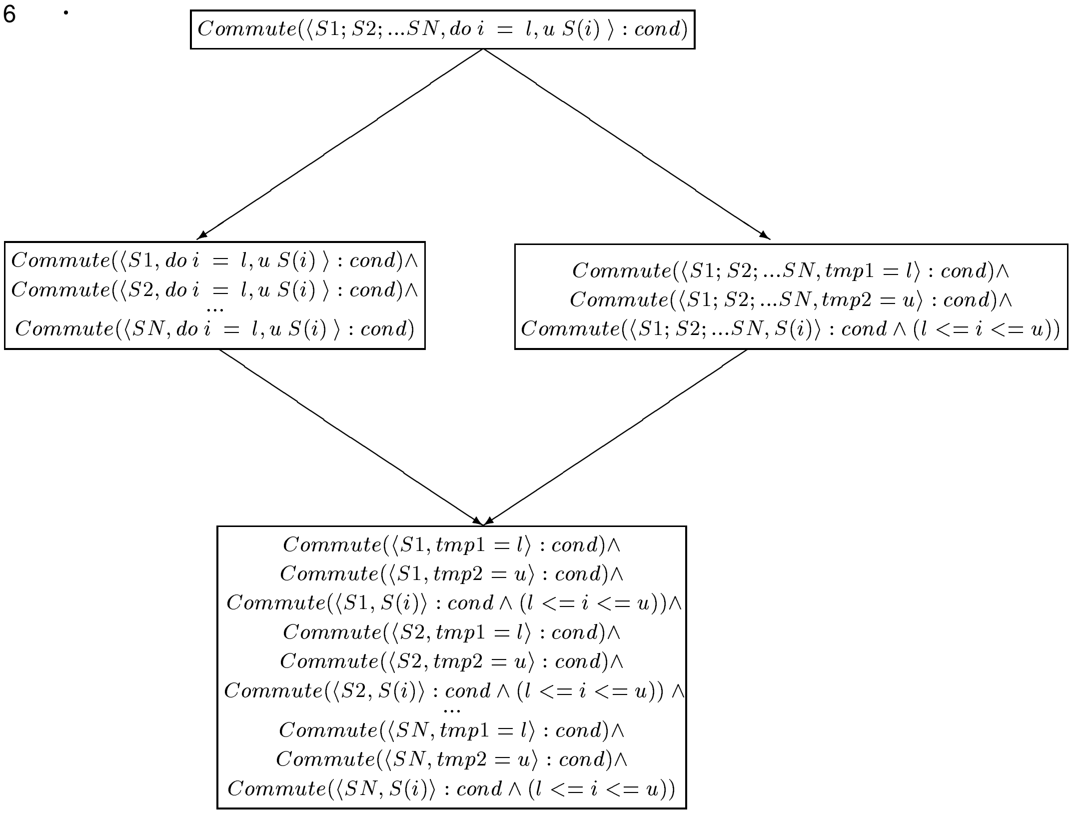


Fig. 13. Recursive Simplification Rules and the Church-Rosser Property

constraints on free variables in the programs[2], and a list of variables that are live at the end of execution of the programs.

If the programs $\{pgm_1; pgm_2\}$ and $\{pgm_2; pgm_1\}$ are *Simple*, it invokes the symbolic analysis and comparison engine. Otherwise, if pgm_1 is not *Simple*, it is simplified using the rules shown in Table I(b). If pgm_1 is *Simple*, pgm_2 is simplified (to keep the code short, we accomplish this by invoking *Commute* recursively with the argument order reversed). If both pgm_1 and pgm_2 are *Simple* but their composition is not, we give up and return false conservatively. It is possible to handle this last case in a more elaborate fashion, but we have not needed this for our applications.

4.5 Relationship with Dependence Analysis

We now show that fractal symbolic analysis is strictly more powerful than dependence analysis, provided that the symbolic analysis engine is adequate. It is obvious that if there are no dependences between programs pgm_1 and pgm_2, these programs must commute in the sense that $\{pgm_1; pgm_2\}$ must be equal to $\{pgm_2; pgm_1\}$. What is not obvious is that a fractal symbolic analyzer can actually prove this equality. We show that any adequate symbolic analysis engine can accomplish this.

[2]See Table I for examples of constraints on free variables.

```
Commute(pgm1,pgm2,bindings,liveVars) {
  if (Simple({pgm1; pgm2}) ∧ Simple({pgm2; pgm1})) then
    return SymbolicallyEqual?({pgm1;pgm2},{pgm2;pgm1}, bindings,liveVars)
  elseif (! Simple(pgm1)) then
    //implementation of Table 1(b)
    case (pgm1)
    {
      //statement composition
        ⟨pgm1';pgm1''⟩ →
          return{ Commute(pgm1',pgm2,bindings,liveVars)
                ∧ Commute(pgm1'',pgm2,bindings,liveVars)}

      //conditional
        ⟨if pred then pgm1' else pgm1''⟩ →
          return {Commute(pgm1',pgm2,bindings||pred,liveVars)
                ∧ Commute(pgm1'',pgm2,bindings||¬pred,liveVars)
                ∧ Commute(tmp = pred,pgm2,bindings||¬pred,liveVars)}
    //loop
        ⟨do i = l, u pgm1'(i)⟩ →
          return {Commute(pgm1'(i),pgm2,bindings||∀i.l ≤ i ≤ u,liveVars)
                ∧ Commute(tmp = l,pgm2,bindings,liveVars)
                ∧ Commute(tmp = u,pgm2,bindings,liveVars)}
    }
  elseif (! Simple(pgm2)) then
    Commute(pgm2,pgm1,bindings,liveVars)
  else return false;
}
```

Fig. 14. The *Commute* function

4.5.1 *The* Independent *predicate.* First, we establish an analog of the recursive simplification rules of Table I(b) for dependence analysis. Let pgm_1 and pgm_2 be programs. We will say that these programs are *Independent* if there are no flow/anti/output dependences between these programs. More formally, we have the following definition.

DEFINITION 3. *Let pgm$_1$ and pgm$_2$ be programs, and C be a set of constraints on the free variables of these programs. Let R_1,W_1,R_2, and W_2 be the set of locations read and written by pgm$_1$ and pgm$_2$ respectively, given C. We will write $Independent(\langle pgm_1, pgm_2\rangle : C)$ if, given C, we can show that*

- $W_1 \cap R_2 = \phi$ *(no flow dependence),*
- $R_1 \cap W_2 = \phi$ *(no anti-dependence), and*
- $W_1 \cap W_2 = \phi$ *(no output dependence).*

Lemma 2 establishes an analog of the recursive simplification rules of Table I(b) for the *Independent* predicate. Note that unlike the rules for the *Commute* predicate, the left and right hand sides of these rules are related by the if and only if operator.

LEMMA 2. *The* Independent *predicate satisfies the following properties.*

$$
\begin{aligned}
(a) Independent(\langle \texttt{S1;S2;...;SN}, \texttt{pgm}_2\rangle : cond) \;\Leftrightarrow\;& Independent(\langle S1, pgm_2\rangle : cond) \wedge \\
& Independent(\langle S2, pgm_2\rangle : cond) \wedge \\
& \ldots \\
& Independent(\langle SN, pgm_2\rangle : cond)
\end{aligned}
$$

$$
\begin{aligned}
(b) Independent(\langle \begin{array}{l}\texttt{do i = l,u}\\ \quad\texttt{S1(i);}\end{array}, \texttt{pgm}_2\rangle : cond) \;\Leftrightarrow\;& Independent(\langle tmp1 = l, pgm_2\rangle : cond) \wedge \\
& Independent(\langle tmp2 = u, pgm_2\rangle : cond) \wedge \\
& Independent(\langle S1(i), pgm_2\rangle : cond \wedge l <= i <= u)
\end{aligned}
$$

$$
\begin{aligned}
(c) Independent(\langle \begin{array}{l}\texttt{if (pred)}\\ \texttt{then S1;}\\ \texttt{else S2;}\end{array}, \texttt{pgm}_2\rangle : cond) \;\Leftrightarrow\;& Independent(\langle tmp = pred, pgm_2\rangle : cond) \wedge \\
& Independent(\langle S1, pgm_2\rangle : cond \wedge pred) \wedge \\
& Independent(\langle S2, pgm_2\rangle : cond \wedge \neg pred)
\end{aligned}
$$

PROOF. Follows trivially from the definition of the *Independent* predicate, and the fact that the read and write sets of a program are computed by taking the union of the read and write sets of the assignment statements contained in that program [Wolfe 1995]. □

4.5.2 *Fractal symbolic analysis is more powerful than dependence analysis*

THEOREM 3. $Independent(\langle pgm_1, pgm_2\rangle : C) \Rightarrow Commute(\langle pgm_1, pgm_2\rangle : C)$, *provided the symbolic analysis engine is adequate*[3].

PROOF. Let $C_0 \equiv Commute(\langle pgm_1, pgm_2\rangle : C) \rightarrow C_1 \rightarrow \ldots \rightarrow C_n$ be the sequence of conjunctions of commute conditions generated by recursive simplification. The final condition C_n is a conjunction of terms of the form $Commute(\langle p_1, p_2\rangle : C')$ where p_1 and p_2 are *Simple*.

Using Lemma 2, we can generate a parallel sequence of the form $I_0 \equiv Independent(\langle pgm_1, pgm_2\rangle : C) \rightarrow I_1 \rightarrow \ldots \rightarrow I_n$. If C_{j+1} is generated from C_j by applying some rule in Table I(b), the analogue of that rule from Lemma 2 is applied to generate I_{j+1} from I_j. From Lemma 2, every independence condition in I_n is true.

Putting C_n and I_n together, we see that C_n is a conjunction of terms of the form $Commute(\langle p_1, p_2\rangle : C')$ where (i) p_1 and p_2 are *Simple*, and (ii) $Independent(\langle p_1, p_2\rangle : C')$. By assumption, the symbolic analysis engine is adequate, so it will return true for each invocation $Commute(\langle p_1, p_2\rangle : C')$. Therefore, the fractal symbolic analyzer can deduce that $Commute(\langle pgm_1, pgm_2\rangle : C)$ is true. □

To show that fractal symbolic analysis with an adequate symbolic analysis engine is strictly more powerful than dependence analysis, it is sufficient to observe that the first challenge problem in Section 2 cannot be solved using dependence analysis alone, but that it can be solved by fractal symbolic analysis even if the symbolic analysis engine handles only straight-line code.

[3]Without loss of generality, we can assume that all variables are live at the end of $\{pgm_1; pgm_2\}$ and $\{pgm_2; pgm_1\}$.

```
dummy:
   do i = 1,n
      sum = sum + A(i)
swap:
   t = x
   x = y
   y = t
compute:
   x = x*x
   y = y*y
```

(a) {dummy;swap;compute }

```
compute:
   x = x*x
   y = y*y
dummy:
   do i = 1,n
      sum = sum + A(i)
swap:
   t = x
   x = y
   y = t
```

(b) {compute;dummy;swap}

Fig. 15. Simplifying statement sequences

4.6 Discussion

The simplification rules of Table I are reasonable, but there are certainly other possibilities. In particular, the rule for simplifying statement sequences can be applied in a coarse-grain manner rather than in a fine-grain way, as discussed earlier. Coarse-grain application gives greater accuracy, as the program in Figure 15 shows. The `compute` statement sequence in this figure clearly commutes with the `dummy` and `swap` statement sequences. If we assume that do-loops are not *Simple*, we must simplify the statement sequence consisting of the `dummy` and `swap` statement sequences. A fine-grain application of the rule for simplifying statement sequences requires us to test whether the `compute` statement sequence commutes with the `dummy` loop and with each of the statements in the `swap` sequence. Clearly, it does not, so we would disallow the transformation. A coarse-grain application of the rule for statement sequences would succeed in this case because the statements in the `swap` sequence collectively commute with the `compute` sequence.

Unfortunately, the rule for simplifying statement sequences can be applied in a coarse-grain manner in multiple ways. Fine-grain application is simpler; furthermore, for our test problems, we have not needed the flexibility that coarse-grain application affords.

The transformations listed in Table I comprise most of the transformations used routinely in modern restructuring compilers, other than scalar or array expansion [Wolfe 1995]. Expansion is a *data* transformation which does not change the order in which computations are done; therefore, we do not know any useful way to apply *fractal* symbolic analysis to proving the correctness of expansion.

5. SYMBOLIC ANALYSIS AND COMPARISON

We now present our symbolic analysis and comparison engine, and characterize programs which can be analyzed directly by this engine.

5.1 Simple programs

Our symbolic analyzer can directly analyze programs that have the following characteristics.

DEFINITION 4. *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.*

For example, the programs of Figure 5(a,b) do not satisfy these conditions because among other things, the subscript `p(j)` in array reference `A(p(j))` is neither a loop

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

Fig. 16. Guarded Symbolic Expressions

constant nor an affine function of the loop index variable `j`. On the other hand, the programs of Figure 9(a,b) satisfy these conditions. The array subscripts `j1` and `p(j1)` are symbolic constants in these programs since there are no assignments to either `j1` or array `p`, while array subscript `i` is an affine function of the surrounding loop index. Furthermore, the loop does not have loop-carried dependences.

It is useful to understand which programs are not *simple*.

LEMMA 3. *Suppose* pgm *is not* simple*. One or more of the following assertions must be true.*

- *There is an array index or loop bound involving a variable which is assigned to by an assignment statement in* pgm.
- *There is an array index or loop bound which is a non-affine function of loop variables and symbolic constants.*
- *There is a conditional whose predicate involves a variable which is assigned to by an assignment statement in* pgm.
- *There is a conditional whose predicate is not a conjunction or disjunction of affine inequalities of loop variables and symbolic constants.*
- *There is a loop with a loop-carried dependence.*

PROOF. By negating the conditions in Definition 4. □

The conditions in Definition 4 guarantee that at most one loop iteration will write to any array location even if a loop nest writes to an array section that is unbounded at compile-time. Since array subscripts are affine functions of loop variables and constants, this ensures that array values can be expressed by a finite expression called a guarded symbolic expression (or GSE for short) which contains symbolic expressions that hold for affinely constrained portions of the array as shown in Figure 16. An example of a GSE was given earlier in Figure 10.

It is important to remember that these conditions are not required of input programs; rather, the *Commute* function recursively simplifies its program parameters until these conditions are met. In other words, these conditions must be met by programs S_n and T_n in Figure 2, not by programs S and T. Obviously, these conditions would be different if we use a different symbolic analysis engine, such as one that can handle reductions.

LEMMA 4. *Any symbolic analysis engine that can analyze* simple *programs is adequate.*

PROOF. We show that such a symbolic analysis engine satisfies the three conditions of Definition 1.

(1) Consider a program consisting of a single assignment statement. Any array index must be a symbolic constant. Furthermore, there are no loops or conditionals. Therefore, such a program is *simple*.

```
SymbolicallyEqual?(p1,p2,bindings,live_vars) {
  live1 = set of live modified variables in p1
  live2 = set of live modified variables in p2
  if(live1 ≠ live2)
    return false

  for each a(k⃗) in live1 {
    tree1 = Build_Expr_Tree(p1,a(k⃗),∅)
    tree2 = Build_Expr_Tree(p2,a(k⃗),∅)

    gse1 = Build_GSE(tree1,bindings)
    gse2 = Build_GSE(tree2,bindings)

    if(¬ Compare_GSEs(gse1,gse2))
      return false
  }
  return true
}
```

Fig. 17. Symbolic Analysis and Comparison of Simple Programs

(2) Consider the cases in Lemma 3. In every case, it is trivial to see that a program that contains *pgm* as a sub-program is itself not *simple*.

(3) Suppose pgm_1 and pgm_2 are *simple*, and that there is no dependence between them. Consider $\{pgm_1; pgm_2\}$. Since properties like affine-ness etc. are preserved under composition, it is easy to see that if this program is not *simple*, it must be the case that there is an array index, loop bound or predicate in pgm_1 which involves a variable that is assigned to in pgm_2 or vice versa. However, in that case, there is a dependence from pgm_1 to pgm_2, which is a contradiction. Therefore, $\{pgm_1; pgm_2\}$ must be simple. A similar arguments holds for $\{pgm_2; pgm_1\}$.

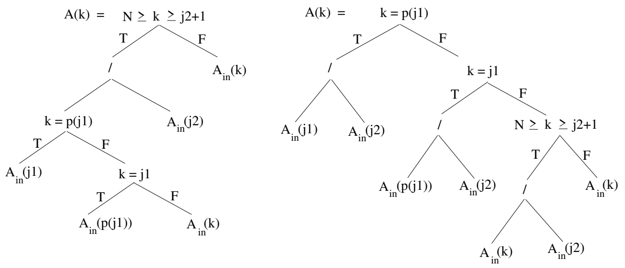
□

Lemma 4 ensures that the results of Section 4 are valid for our implementation of fractal symbolic analysis. In particular, our analysis technique is more powerful than dependence analysis in the sense of Theorem 3. To get some perspective in our discussions, we will sometimes refer to an assertion as being *Independently true* if both dependence analysis and our fractal symbolic analysis can prove that assertion; if only our fractal symbolic analysis can prove it, we will refer to it as being *Symbolically true*.

Figure 17 provides a high-level overview of our symbolic analysis and comparison algorithm for programs that are *simple*. If the GSE's from the two programs for every live and modified variable are equal, the two programs are declared to be equal. The generation of GSE's is done in two phases. First, we build *conditional expression trees* for each variable as described in Section 5.2 to represent the output value of that variable as a function of inputs. Then, these trees are normalized by pushing arithmetic operators below predicates, and the GSE is read off directly from these normalized expression trees, as described in Section 5.3.

5.2 Generation of Conditional Expression Trees

Figure 18 shows the conditional expression trees for the programs of Figure 9(a,b). The interior nodes of the tree are predicates and operators while the leaves of the tree are scalars and array references. Since simplified programs have only straight-line code and DO-ALL loops, the construction of these trees is straight-forward, as shown in Figure 19. Procedure



(a) Tree for Figure 9(a) (b) Tree for Figure 9(b)

Fig. 18. Conditional Expression Trees for Running Example

Build_Expr_Tree($stmt$,$expr$,$bindings$) {
  case ($expr$)
  {
    Op(op,$expr_1$,...,$expr_n$) :
      return Op(op,
          Build_Expr_Tree($stmt$,$expr_1$,$bindings$),...,
          Build_Expr_Tree($stmt$,$expr_n$,$bindings$))

    Cond($pred$,$expr_t$,$expr_f$) :
      return Cond($pred$,
          Build_Expr_Tree($stmt$,$expr_t$,$bindings$),
          Build_Expr_Tree($stmt$,$expr_f$,$bindings$))

    A($\vec{k}$) :
      case ($stmt$)
      {
        $\langle$A'(c) = $expr_1$$\rangle$: //constant subscript
          if (A $=$ A') **then**
            return Cond($bindings \| \vec{k} = c$,
                $expr_1$,A($\vec{k}$))
          else
            return A($\vec{k}$)

        $\langle$A'($T \cdot \vec{i} + c$) = $expr_1(\vec{i})$$\rangle$: //general affine subscript
          if (A $=$ A') **then**
            return Cond($bindings \| \vec{k} = T \cdot \vec{i} + c$,
                $expr_1(T^{-1} \cdot (\vec{k} - c))$,A($\vec{k}$))
          else
            return A($\vec{k}$)

        $\langle stmt_1;stmt_2 \rangle$:
          return Build_Expr_Tree($stmt_1$, Build_Expr_Tree($stmt_2$,$expr$,$bindings$), $bindings$)

        $\langle$`if` $pred$ `then` $stmt_1$ `else` $stmt_2$$\rangle$:
          return Cond($bindings \| pred$,
            Build_Expr_Tree($stmt_1$,$expr$,$bindings$),
            Build_Expr_Tree($stmt_2$,$expr$,$bindings$))

        $\langle$`do` $i_k = l_k, u_k$ $stmt_1$$\rangle$:
          return Build_Expr_Tree($stmt_1$,$expr$,$bindings \| \exists i_k . l_k \leq i_k \leq u_k$)
      }
  }
}

Fig. 19. Expression Tree Generation

```
Build_GSE(tree,bindings) {
  return Flatten(Normalize(tree),bindings)
}


Normalize(tree) {
  case (tree) {
    Op(op,tree_1,...,tree_n) :
      return Exchange(Op(op,Normalize(tree_1),...,,Normalize(tree_n)));

    Cond(pred,tree_t,tree_f) :
      return Cond(pred,Normalize(tree_t),Normalize(tree_f));

    default:return tree;
  }
}


Exchange(tree) {
  case (tree) {
    Op(op,tree_1,...,Cond(pred,tree_t_i,tree_f_i),...,tree_n) :
      return Cond(pred,
               Exchange(Op(op,tree_1,...,tree_t_i,...,tree_n)),
               Exchange(Op(op,tree_1,...,tree_f_i,...,tree_n)));

    Op(op,tree_1,...,tree_n) : //tree_1 ...tree_n not conditionals
      return tree;
  }
}


Flatten(tree,guard) {
  if (non-empty(guard)) then
    case (tree) {
      Cond(pred,tree_t,tree_f) :
        return Flatten(tree_t,guard ∧ pred) ⋃
               Flatten(tree_f,guard ∧ ¬pred);

      default:   return {(guard → expr)};
    }
  else
    return ∅;
}
```

Fig. 20. From Expression Trees to GSE's

Build_Expr_Tree processes the statements of a program in reverse order, determining at each step the tree corresponding to relevant output data in terms of input data and linking these together to produce the final result. The first parameter *stmt* is a statement, the second parameter *tree* is an expression (such as an array reference), and the third parameter *bindings* is a set of constraints on variables. The procedure computes the value of the expression *tree* as a function of the values of variables before statement *stmt* is executed, using the constraints in parameter *bindings* to permit simplification of this function by eliminating impossible cases. For example, to compute the tree shown in Figure 18(a), this procedure is passed the program of Figure 9(a) as the first parameter, the expression `A(k)` as the second parameter, and the constraint $j1 > j2$ as the third parameter.

5.3 Generating Guarded Symbolic Expressions

In general, conditional expression trees contain interleaved affine constraints and arithmetic expressions, as shown in Figure 18(a). To generate GSE's, it is convenient to separate affine constraints from arithmetic expressions (Figure 18(b) is a conditional expression tree which exhibits this separation fortuitously). We accomplish this separation by repeated

```
Compare_GSEs(gse1,gse2) {
  for each (guard1,expr1) in gse1 {
    for each (guard2,expr2) in gse2 {
      if (non-empty(guard1 ∧ guard2))
        if (expr1 ≠ expr2) // symbolic comparison
          return false
    }
  }

  return true
}
```

Fig. 21. Comparison of GSE's

application of the following transformation.

$\mathrm{Op}(op,tree_1,...,\mathrm{Cond}(pred,tree\_t_i,tree\_f_i),...,tree_n) \Rightarrow$
$\mathrm{Cond}(pred,\mathrm{Op}(op,tree_1,...,tree\_t_i,...,tree_n), \mathrm{Op}(op,tree_1,...,tree\_f_i,...,tree_n))$

After the separation is done, guards are generated by flattening the affine constraints at the top of the expression tree, and the corresponding arithmetic expressions are taken from the subtrees beneath these predicates, as is shown in Figure 20. In this code, the `Normalize` function is responsible for bubbling conditionals to the top of the expression tree. It accomplishes this in a bottom-up way — after normalizing the sub-trees of the top-level node, it invokes the `Exchange` method which repeatedly applies the transformation shown above to push the operator at the topmost level of its parameter `tree` past any conditionals. Once this is done, the `Flatten` function generates the GSE. This function checks that its parameter `guard`, which is the conjunction of affine equalities and inequalities, defines a non-empty region; if so, it walks down the normalized expression tree and generates the GSE.

5.4 Comparison of Guarded Symbolic Expressions

Finally, Figure 21 illustrates the comparison of two guarded symbolic expressions. There are two steps to this comparison. First, we must compare each pair of affine guards of the two guarded symbolic expressions. Second, for any two guards that potentially intersect, we must compare the corresponding symbolic expressions for equality. It is during this comparison that we use any algebraic laws that are obeyed by the operators in the two expressions. If every comparison returns true, then the guarded symbolic expressions are declared to be equal. The validity of this conclusion follows from the following argument. Each guard specifies some region of the index space of the array in question, and the union of these regions in a guarded symbolic expression is equal to the entire index space of that array. If the values in the two guarded symbolic expressions are identical whenever their guards intersect, the two array values are obviously equal.

For comparison of affine guards, we may employ an integer programming tool such as the Omega Library [Pugh 1992]. If the tool proves that a pair of affine guards do not intersect, no comparison of the corresponding arithmetic expressions needs to be performed. On the other hand, if the guards do intersect, the expressions must be compared for equality. This comparison is done symbolically, using any algebraic laws that are obeyed by the operators in the expressions, such as commutativity and associativity. In our current implementation, we just check for *syntactic* equality. This is sufficient for restructuring LU factorization, as we will see in Section 6.

```
        i = 1
     A(i) = 2
```

Fig. 22. A program that is not *simple*

5.5 Discussion

Definition 4 of *simple* programs rules out many programs that are not hard to analyze symbolically, such as the program in Figure 22. It is easy to make the symbolic analysis engine more powerful to handle programs such as this one; the point of the engine described here is that it is the simplest one we know of that can address all the analysis problems that arise in restructuring LU with pivoting.

It is important to realize that the framework of fractal symbolic analysis does not rely in any way on the interpretation of function symbols in arithmetic expressions. If arithmetic operators such as addition can be assumed to obey algebraic laws such as commutativity and associativity, these laws can be used in proving equality of expressions in Figure 21. If the use of these properties to restructure programs may change the numerical properties of the algorithm, only syntactic equality is used in proving expression equality. This is the only place in the entire framework where algebraic properties of operators are used, and the choice of whether to use these properties is under the control of the compiler writer.

6. LU WITH PIVOTING

Fractal symbolic analysis was developed for use in an ongoing project on optimizing the cache behavior of dense numerical linear algebra programs. LU factorization with partial pivoting is a key routine in this application area since it is used to solve systems of linear equations of the form `Ax = b`. Figure 23(a) shows the canonical version of LU factorization with pivoting that appears in the literature [Golub and Loan 1996]. In iteration `j` of the outer loop, row `j` of matrix `A` is swapped with some row `p(j)` (where `p(j)` $\geq$ `j`), computations are performed on column `j`, and a portion of the matrix to the right of this column is updated. This version of LU factorization with pivoting is called a *right-looking* code because the updates are performed on columns to the right of the current column `j`. There is also a *left-looking* version, shown in Figure 23(b). In this version, the swaps and updates to a column are performed "lazily" — that is, the swaps and updates to column `j` are performed when column `j` becomes current, and then the computations on that column are performed. It is known that the two versions of LU factorization with pivoting compute identical values even if addition and multiplication are not associative [Golub and Loan 1996].

LU factorization with pivoting poses a number of challenges for restructuring compilers.

(1) Cache-optimized versions of LU factorization can be found in the LAPACK library [Anderson et al. 1995]. These *blocked codes* are too complex to be reproduced here, but on machines with memory hierarchies, they perform much better than the *point versions* shown in Figure 23.

Given point-wise LU factorization with pivoting, can a compiler automatically generate a cache-optimized version by blocking the code? We address this problem in Section 6.1.

How does the performance of the compiler-optimized code compare with that of hand-blocked code? We discuss experimental results in Section 6.2.

```
do j = 1, N
  // 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
  do k = 1, N
    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)
  // Update sub-matrix
  do k = j+1, N
    do i = j+1, N
      A(i,k) = A(i,k) - A(i,j)*A(j,k)
```

(a) Right-looking Code

```
do j = 1, N
  // Apply delayed swaps to current column
  do k = 1, j-1
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp
  // Apply delayed updates
  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 row
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i
  // Swap all 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 Code

Fig. 23. LU Factorization with Pivoting

(2) Right-looking and left-looking versions of LU factorization are extensionally equal even if addition and multiplication do not obey the usual algebraic laws. Can a compiler transform right-looking LU with pivoting to the left-looking version, and vice versa? This problem was addressed by us in an earlier paper [Mateev et al. 2000].

Neither dependence analysis nor standard symbolic analysis are adequate to address these challenges. We now show how fractal symbolic analysis solves the problem of blocking LU with pivoting.

6.1 Automatic Blocking of LU Factorization

To obtain code competitive with LAPACK code, Carr and Lehoucq suggest carrying out the following sequence of restructuring transformations [Carr and Lehoucq 1997]. Descriptions of these transformations can be found in any text-book on restructuring compilers such as Wolfe [Wolfe 1995].

(1) Stripmine the outer loop to expose block column operations.
(2) Index-set-split the expensive update operation to separate computation outside the current block column from computation inside the current block-column.
(3) Distribute the inner of the stripmined loops to isolate the update to columns to the right of the current block column.
(4) Tile the update to the columns to the right of the current block column.

The first two steps, stripmining and index-set-splitting, are trivially legal as they do not reorder any computation. The third step, loop distribution, is shown in Figures 24(a,b), and is not necessarily legal. If legality is checked using dependence analysis, the compiler will declare the distribution illegal if there is a dependence from an instance `B2(m)` to an instance `B1(l)` where `l` $>$ `m`. In fact, such a dependence exists in our program; for example, both `B2(j)` and `B1(j+1)` read and write to `A(m+1,jB+B..N)`. Therefore,

```
  do jB = 1, N, B
    do j = jB, jB+B-1
B1(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
      do k = 1, N
        tmp = A(j,k)
        A(j,k) = A(p(j),k)
        A(p(j),k) = tmp
      // Scale column
      do i = j+1, N
        A(i,j) = A(i,j) / A(j,j)
      // In-Column Update
      do k = j+1, jB+B-1
        do i = j+1, N
          A(i,k) = A(i,k) - A(i,j)*A(j,k)
B2(j):
      // Right-Looking Update
      do k = jB+B, N
        do i = j+1, N
          A(i,k) = A(i,k) - A(i,j)*A(j,k)
```

(a) Before Loop Distribution

```
  do jB = 1, N, B
    do j = jB, jB+B-1
B1(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
      do k = 1, N
        tmp = A(j,k)
        A(j,k) = A(p(j),k)
        A(p(j),k) = tmp
      // Scale column
      do i = j+1, N
        A(i,j) = A(i,j) / A(j,j)
      // In-Column Update
      do k = j+1, jB+B-1
        do i = j+1, N
          A(i,k) = A(i,k) - A(i,j)*A(j,k)
    // Distributed Loop
    do j = jB, jB+B-1
B2(j):
      // Right-Looking Update
      do k = jB+B, N
        do i = j+1, N
          A(i,k) = A(i,k) - A(i,j)*A(j,k)
```

(b) After Loop Distribution

```
B1.a(l): p(l) = l
B1.b(l): do i = l+1, N
           if abs(A(i,l)) > abs(A(p(l),l))
             p(l) = i
B1.c(l): do k = 1, N
           tmp = A(l,k)
           A(l,k) = A(p(l),k)
           A(p(l),k) = tmp
B1.d(l): do i = l+1, N
           A(i,l) = A(i,l) / A(l,l)
B1.e(l): do k = l+1, jB+B-1
           do i = l+1, N
             A(i,k) = A(i,k) - A(i,l)*A(l,k)
B2(m):   do k = jB+B, N
           do i = m+1, N
             A(i,k) = A(i,m) - A(i,m)*A(m,k)
```

(c) {`B1(l); B2(m)`}

```
B2(m):   do k = jB+B, N
           do i = m+1, N
             A(i,k) = A(i,m) - A(i,m)*A(m,k)
B1.a(l): p(l) = l
B1.b(l): do i = l+1, N
           if abs(A(i,l)) > abs(A(p(l),l))
             p(l) = i
B1.c(l): do k = 1, N
           tmp = A(l,k)
           A(l,k) = A(p(l),k)
           A(p(l),k) = tmp
B1.d(l): do i = l+1, N
           A(i,l) = A(i,l) / A(l,l)
B1.e(l): do k = l+1, jB+B-1
           do i = l+1, N
             A(i,k) = A(i,k) - A(i,l)*A(l,k)
```

(d) {`B2(m); B1(l)`}

```
B1.c(l): do k = 1, N
           tmp = A(l,k)
           A(l,k) = A(p(l),k)
           A(p(l),k) = tmp
B2(m):   do k = jB+B, N
           do i = m+1, N
             A(i,k) = A(i,m) - A(i,m)*A(m,k)
```

(e) {`B1.c(l); B2(m)`}

```
B2(m):   do k = jB+B, N
           do i = m+1, N
             A(i,k) = A(i,m) - A(i,m)*A(m,k)
B1.c(l): do k = 1, N
           tmp = A(l,k)
           A(l,k) = A(p(l),k)
           A(p(l),k) = tmp
```

(f) {`B2(m); B1.c(l)`}

Fig. 24. Loop Distribution in LU: Fractal Symbolic Analysis

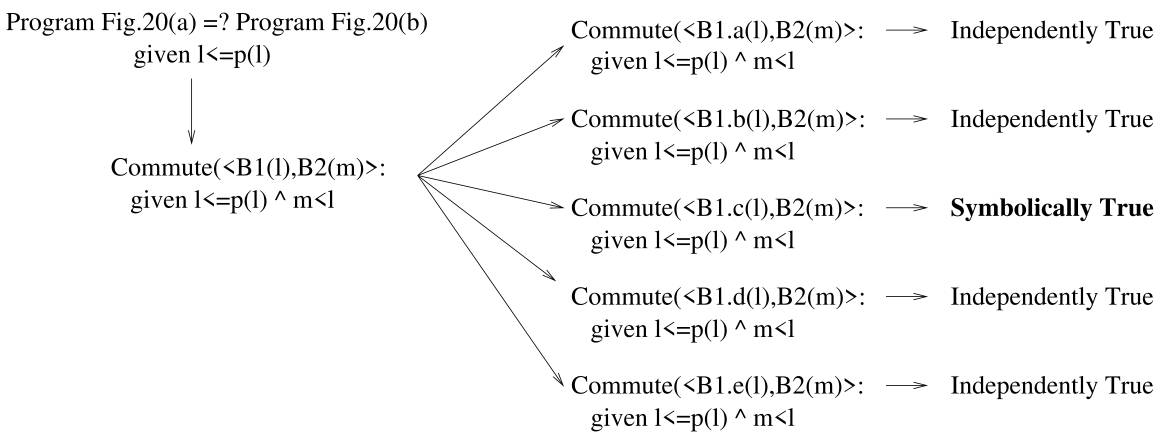


Fig. 25. Commute Calls for LU

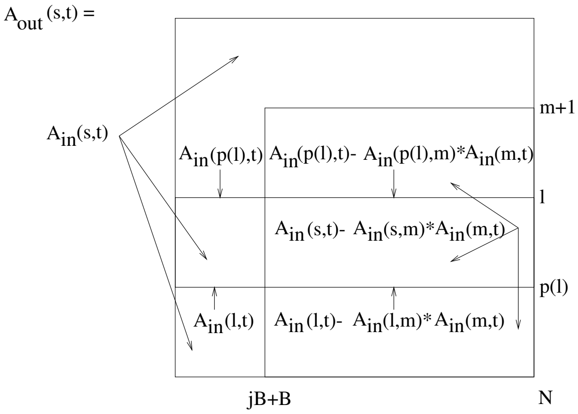
a compiler that relies on dependence analysis cannot block LU with pivoting using the transformation strategy of Carr and Lehoucq. Symbolic analysis of LU with pivoting is too difficult. Carr and Lehoucq suggest that a compiler may be endowed with application-specific information to recognize that the distribution is legal. The disadvantage of this solution is that it is not a general-purpose solution, and it is likely that any such approach would have to be based on pattern-matching, which is notoriously fragile.

Fractal symbolic analysis solves this problem elegantly. A high level view of the recursive simplification steps is shown in Figure 25. To verify the legality of the loop distribution step of Figure 24(a,b), our compiler consults Table I(a) and determines that it must check if $\{$`B1(l);B2(m)`$\}$ is extensionally equal to $\{$`B2(m);B1(l)`$\}$ where `jB` $\leq$ `m` $<$ `l` $\leq$ `jB+B-1` (these two programs are shown in Figure 24(c,d)). These programs are simpler than the full code which performs LU with factorization, but they are still not simple enough because the loop that computes the pivot in `B1(l)` is a recurrence and cannot be analyzed symbolically by the analysis engine described in Section 5. Therefore, these programs are simplified again using the rule for statement sequences in Table I(b). In Figure 24(c,d), the five statements in the block `B1(l)` are labeled `B1.a(l)...` `B1.e(l)`; the compiler must check whether `B2(m)` commutes with each of these sub-blocks. None of the sub-blocks other than `B1.c(l)` touch locations that are also touched by `B2.m(l)`; therefore dependence analysis is adequate for all sub-blocks other than `B1.c(l)` (a small detail is that the analysis of whether $\{$`B1.b(l);B2(m)`$\}$ is equal to $\{$`B2(m);B1.b(l)`$\}$ requires an additional step of simplification to eliminate the recurrence in `B1.b(l)`).

The remaining problem is to show that `B1.c(l)` commutes with `B2(m)`. At this point, these programs are "simple enough", and the `SymbolicallyEqual?` method in Figure 17 is invoked to establish equality of the simplified programs. In fact, they are quite similar to the running example of Section 2. The only live, altered variable in either program is the array `A`, and the `SymbolicallyEqual?` method generates guarded symbolic expressions for `A` from each program. Both GSE's generated from Figure 24(e,f) contain six guards, as shown in Figure 26(a); a pictorial representation of the GSE's is shown in Figure 26(b). These regions correspond to the following computations:

$$A_{out}(s,t) = \begin{cases} (s = l) \wedge (1 \le t \le jB + B - 1) & \rightarrow & A_{in}(p(l), t) \\ (s = l) \wedge (jB + B \le t \le N) & \rightarrow & A_{in}(p(l), t) - A_{in}(p(l), m) * A_{in}(m, t) \\ (s = p(l)) \wedge (1 \le t \le jB + B - 1) & \rightarrow & A_{in}(l, t) \\ (s = p(l)) \wedge (jB + B \le t \le N) & \rightarrow & A_{in}(l, t) - A_{in}(l, m) * A_{in}(m, t) \\ (m + 1 \le s \le N) \wedge (jB + B \le t \le N) \wedge (s \ne l) \wedge (s \ne p(l)) & \rightarrow & A_{in}(s, t) - A_{in}(s, m) * A_{in}(m, t) \\ else & \rightarrow & A_{in}(s, t) \end{cases}$$

(a) GSE for A_{out} in Figures 24(e,f)



(b) Pictorial representation of GSE in (a)

Fig. 26. LU with pivoting: Guarded Symbolic Expression for A_{out}

(1) `A(l,1..jB+B-1)`: prefix of row `l`, written by the swap `B1.c(l)` but not by the update `B2(m)`

(2) `A(l,jB+B..N)`: suffix of row `l`, written by the swap `B1.c(l)` and by the update `B2(m)`

(3) `A(p(l),1..jB+B-1)`: prefix of row `p(l)`, written by the swap `B1.c(l)` but not by the update `B2(m)`

(4) `A(p(l),jB+B..N)`: suffix of row `p(l)`, written by the swap `B1.c(l)` and by the update `B2(m)`

(5) the submatrix `A(m+1..N,jB+B..N)` excluding rows `l` and `p(l)`: these locations are written by the update `B2(m)` but are left untouched by the swap `B1.c(l)`

(6) the rest of the matrix: these elements are left untouched by both the swap and the update

The GSE's generated from the programs of Figures 24(e,f) are syntactically identical. In our implementation, `Compare_GSEs` is invoked to generate the 36 pairwise intersections, and the Omega library [Pugh 1992] is used to test non-emptiness of these regions. Only six intersections are non-empty (the six regions shown in Figure 26), and the corresponding symbolic expressions are syntactically identical in each case. Thus, the compiler is able to demonstrate the equality of the simplified programs and, therefore, the equality of the programs in Figure 24(a,b). Since the symbolic expressions are syntactically equal, it also follows that the restructuring does not change the output of the program even if arithmetic is finite-precision (that is, even if addition and multiplication do not obey the usual algebraic laws).

One important note is that the programs of Figure 24(a,b) are equal only if $p(j) \geq j$. Techniques such as value propagation [Maslov 1995; DeRose 1996] have been developed to perform this type of analysis for indirect array accesses to compute dependences more accurately. It is clear that this information may easily be inferred from the pivot computation in `B1.a` and `B1.b`. In our implementation, this information is passed by the compiler as bindings to the method *Commute* along with the legality conditions in Table I.

With this information, our implementation of fractal symbolic analysis is able to automatically establish the legality of the loop distribution transformation in Figure 24. For this example, our implementation, prototyped in Caml-Light [Leroy 1996], took slightly less than one second. Most of the analysis time is spent on the construction and comparison of guarded symbolic expressions since we have not yet optimized the code for doing this.

6.2 Experimental Results

We now present experimental results that demonstrate the effectiveness of the blocking strategy discussed above. These experiments were conducted on a 300 MHz SGI Octane with a 2 MB L2 cache.

Figure 27 shows the improvement in performance that results from blocking right-looking LU with pivoting as discussed above. The lowest line (labeled `Right-looking LU`) was obtained by running the code generated by the SGI MIPSPro compiler. The MIPSPro compiler uses a sophisticated blocking algorithm for perfectly-nested loop nests, and it uses transformations like loop jamming to convert imperfectly-nested loop nests into perfectly-nested ones where possible. Nevertheless, it uses dependence analysis, so it is not able to block the LU code effectively. Notice that performance starts to drop when the matrix size is about 500x500. Each element of this matrix is a double-word, so a matrix of this size occupies 2 MB storage, which corresponds to the size of the L2 cache as expected. Performance levels off at about 50 MFlops which is a small fraction of the peak performance of this machine.

To see what can be achieved with blocking, we downloaded a hand-blocked version of LU factorization with pivoting from Netlib[4]. This portable version achieved about 450 MFlops. Even better performance (475 MFlops) is obtained by using a version of LAPACK tuned for the SGI Octane. The top two lines in Figure 27 show the performance of these two versions.

To see what performance the MIPSPro compiler might have achieved had it used fractal symbolic analysis, we applied the first three Carr/Lehoucq transformations by hand to the right-looking code of Figure 23. The resulting code was shown in Figure 24(b). If this code is input to the MIPSPro compiler, the compiler is able to block the right-looking update since it is a 3-deep perfectly-nested loop nest, thereby accomplishing the last step of the Carr/Lehoucq blocking sequence on its own. The resulting performance is shown by the line labeled `Distributed update`. Notice that the performance does not drop beyond matrix size 500x500, showing that we have blocked effectively for the L2 cache.

Nevertheless, this code, at 200 MFlops, is still a factor of two slower than the hand-blocked codes. Further experimentation found that the remaining performance gap was due to the SGI compiler's sub-optimal treatment of the right-looking update computation.

[4] http://www.netlib.org

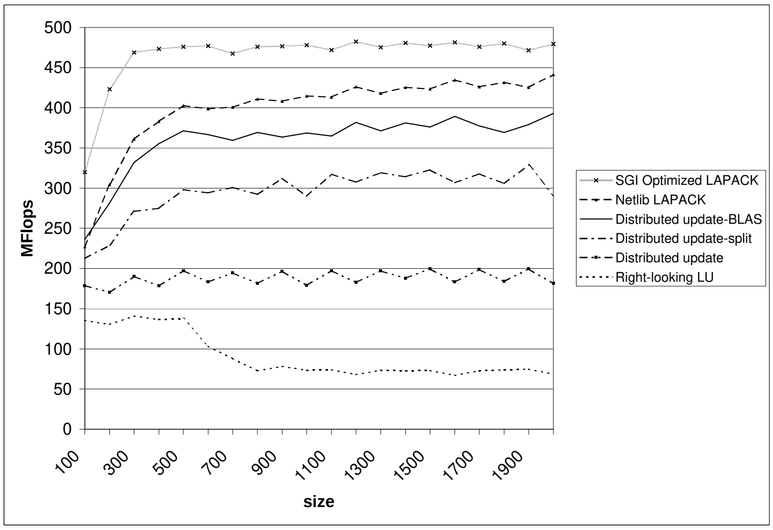


Fig. 27. Experimental Results

Although, the SGI compiler is able to block the update, we conjectured that it might have been confused by the partially triangular loop bounds of the update. When we index-set split the `i` loop by hand to separate the triangular and rectangular portions of the update, the compiler generated substantially faster code, achieving over 300 MFlops. Finally, if we replaced the triangular and rectangular portions of the update with the corresponding BLAS-3 calls (DTRSM and DGEMM) used in LAPACK, the resulting code achieves nearly 400 MFlops and is within 10% of Netlib LAPACK and 20% of the best code in the vendor-supplied library.

These results show that a compiler which uses fractal symbolic analysis should, in principle, be able to restructure LU with pivoting and obtain performance comparable to that of the LAPACK library code. However, this requires improvements in the SGI compiler's ability to optimize perfectly-nested loop nests so that the performance of compiler-generated code for the BLAS becomes comparable to that of hand-written BLAS library code.

7. RELATED WORK

The use of symbolic analysis in compilers has a long history. A simple kind of symbolic analysis called *value numbering* [Aho et al. 1986] and a generalization called *global value numbering* [Reif and Tarjan 1982] are used in some optimizing compilers to identify opportunities for common subexpression elimination and constant propagation, but these techniques are not useful for comparing *different* programs.

Sophisticated symbolic analysis techniques for finding *generalized induction variables* have been developed by Haghighat and Polychronopoulos [Haghighat and Polychronopoulos 1996] and by Rauchwerger and Padua [Rauchwerger and Padua 1999], but their goal is to perform strength-increasing to eliminate loop-carried dependences, thereby enhancing program parallelism. Since this may produce DO-ALL loops from loops with loop-carried dependences, it may be advantageous to preprocess programs in this way before applying fractal symbolic analysis with the symbolic analyzer described in Section 5, since this may eliminate the need for recursive simplification in such programs.

Commutativity analysis [Rinard and Diniz 1997] is a program parallelization technique that uses symbolic analysis to determine if method invocations can be executed concurrently. This approach is based on the insight that a sequence of atomic operations can be executed in parallel if each pair of operations can be shown to commute. Both fractal symbolic analysis and commutativity analysis use symbolic analysis and comparison of programs. However, there is no analog of recursive simplification in commutativity analysis. In particular, the high-level control structure of fractal symbolic analysis, shown in Figure 2, is not present in commutativity analysis. In addition, requiring all operations to commute with each other is too strong a condition for verifying loop transformations.

The algorithm for generating conditional expression trees in Section 5 is reminiscent of backward slicing [Weiser 1984] which is a technique that isolates the portion of a program that may affect the value of a variable at some point in the program. Our algorithm is simpler than the usual algorithms for backward slicing since the programs it must deal with have been simplified beforehand by recursive simplification, an operation that has no analog in backward slicing.

The logic synthesis community has considered a representation of Boolean formulae called *Binary Decision Diagrams* (BDD's) [Bryant 1986]. Conditional expression trees resemble generalized BDDs in which nodes represent affine constraints on variables, rather than boolean variables. Although the manipulation of conditional expression trees does not take much time in our implementation, it is conceivable that this manipulation can be speeded up using techniques from the extensive BDD literature.

Barth and Kozen are exploring the use of purely axiomatic techniques as an alternative to fractal symbolic analysis [Barth and Kozen 2002]. They have used Kleene algebra with tests to prove that the transformations of the two challenge problems in Section 2 are legal. It remains to be seen if this technique can be scaled up to tackle larger kernels like LU with pivoting.

There is also a large body of unrelated work that should not be confused with fractal symbolic analysis.

Fractal symbolic analysis is not related to *value-based dependence analysis* [Feautrier 1991], as discussed in Section 1. In conventional dependence analysis, a dependence is assumed to exist from a statement instance that writes to a variable x to a statement instance that reads from x even if there are intermediate statement instances that write to this variable. For some applications such as array privatization, it is necessary to identify the last write to a location that occured before that location is read by a particular statement instance. Value-based dependence analysis is a more precise variation of dependence analysis which computes this last-write-before-read information. It can be shown that value-based dependence analysis is not adequate to solve the problems with restructuring LU that were discussed in this paper.

Fractal symbolic analysis is also not related to algorithm recognition [Metzger and Wen 2000] or pattern matching in compilers [Wolfe 1995]. We compute symbolic expressions which represent the values of program outputs as functions of their inputs, and determine the equality of these expressions under relevant domain-specific algebraic laws. Therefore, we make no effort to recognize algorithms, nor do we replace one algorithm by another. In principle, algorithm replacement can replace one sorting method with another, which is beyond the scope of fractal symbolic analysis. However, fractal symbolic analysis is more general purpose in the sense that its applicability is not restricted to a certain set of built-in patterns or algorithms; it is also not as easily confused by syntactic clutter or by complex contexts.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced an approximate symbolic analysis called *fractal symbolic analysis*, and used it to solve the long-open problem of restructuring LU with pivoting. If direct symbolic analysis of a program and its transformed version is too difficult, fractal symbolic analysis generates simpler programs whose equality is sufficient (but not in general necessary) for equality of the original programs. Repeated application of the simplification rules is guaranteed to produce programs that are simple enough to be analyzed even by a symbolic analyzer that can only handle straight-line code. We also showed that under some reasonable conditions on the base symbolic analyzer, fractal symbolic analysis is strictly more powerful than dependence analysis.

The work described in this paper can be extended in many ways.

The symbolic analysis engine can be extended to recognize and summarize reductions involving associative arithmetic operations like addition and multiplication, and the symbolic comparison engine can invoke a symbolic algebra tool like Maple [Char et al. 1983] to compare such expressions using the usual algebraic laws. These enhancements might eliminate the need for recursive simplification in some programs, but we do not yet have any applications where this additional power is needed. The finite precision of computer arithmetic means that floating-point addition and multiplication do not necessarily obey the usual algebraic laws, so these laws must be used with care.

The intuition behind fractal symbolic analysis is to view a program transformation as a multi-step process which transforms the initial program incrementally to the final program. In general, there are many multi-step processes that achieve the effect of a given transformation, as discussed in Section 3. Even if we restrict attention to sequences of adjacent transpositions, some sequences may preserve program semantics at every step, while others may take the program through intermediate states in which program semantics is violated. Is it useful to explore an entire space of incremental processes for converting one program to another? If so, how do we manage the search to keep it tractable?

The proof of correctness of the transformation for LU with pivoting discussed in Section 6 required knowing that $p(j) \geq j$. This constraint is easy to deduce, but how does a compiler know in general that this information is useful? One approach is to have the compiler gather as many constraints on variables as it can deduce, and pass them to the fractal symbolic analyzer. An alternative, lazy strategy is to gather only facts that are required for proving the validity of transformations, but it is not clear how such facts can be identified.

Finally, we note that dependence information for loops can be represented abstractly using dependence vectors, cones, polyhedra etc. These representations have been exploited to *synthesize* transformations to optimize performance. 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.

The obvious solution is to compute dependence information, eliminate some of the apparent dependences by performing symbolic analysis, and then use the remaining dependences to synthesize program transformations. Unfortunately, dependences may be too fine-grain for this strategy to be effective. In dependence analysis, granularity of analysis is not an issue because two blocks cannot be independent if they have subblocks that are dependent. Therefore, we can choose as low a level of granularity as we want, so we usually compute dependences between statement instances. In symbolic analysis, two blocks may commute even though they have individual subblocks that do not. Figure 15 shows a simple program that illustrates the problem. The `swap` and `compute` blocks commute, but an analysis that starts with statement-level dependences will not be able to determine this. A non-trivial example is provided by LU with pivoting: the pivot block and the update block must be considered in their entirety to establish the legality of the loop distribution discussed in Section 6.

Another possibility is to use an approach like data shackling [Kodukula et al. 1997] to generate transformations heuristically, and verify correctness using fractal symbolic analysis.

We leave these problems for future work.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques and Tools*, Second ed. Addison Wesley, Reading, MA.

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D., Eds. 1995. *LAPACK Users' Guide. Second Edition*. SIAM, Philadelphia.

BANERJEE, U. 1989. A theory of loop permutations. In *Languages and compilers for parallel computing*. 54–74.

BARTH, A. AND KOZEN, D. 2002. Equational verification of cache blocking in lu decomposition using kleene algebra with tests. Tech. Rep. 2002-1865, Cornell University.

BERNSTEIN, A. 1966. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers EC-15,* 5, 757–763.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. C35,* 8 (Aug.).

CARR, S. AND LEHOUCQ, R. B. 1997. Compiler blockability of dense matrix factorizations. *ACM Transactions on Mathematical Software 23,* 3 (September), 336–361.

CHAR, B., GEDDES, K., AND GONNET, G. 1983. The Maple symbolic computation system. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation) 17,* 3/4 (August/November), 31–42.

COOPER, K. D., KENNEDY, K., AND TORCZON, L. 1986. The impact of interprocedural analysis and optimization in the r^n programming environment. *ACM Trans. Program. Lang. Syst. 8,* 4 (Oct.), 491–523.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1992. *Introduction to Algorithms*. MIT Press.

DEROSE, L. A. 1996. Compiler techniques for MATLAB programs. Technical Report 1956, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. May.

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20,* 1 (February), 23–53.

FEAUTRIER, P. 1992. Some efficient solutions to the affine scheduling problem - part 1: one dimensional time. *International Journal of Parallel Programming*.

GOLUB, G. AND LOAN, C. V. 1996. *Matrix Computations*. The Johns Hopkins University Press.

GUNTER, C. A. 1992. *Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, Cambridge, Massachusetts.

HAGHIGHAT, M. R. AND POLYCHRONOPOULOS, C. D. 1996. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 18,* 4 (July), 477–518.

JOHNSON, S. M. 1963. Generation of permutations by adjacent transposition (in Technical Notes and Short Papers). *Mathematics of Computation 17,* 83 (July), 282–285.

J.W.KLOP. 1980. *Combinatory Reduction Systems*. Mathematisch Centrum, Amsterdam.

KODUKULA, I., AHMED, N., AND PINGALI, K. 1997. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN.

LEROY, X. 1996. The Caml Light system, release 0.71. Documentation and user's manual. Tech. rep., INRIA. March. Available from *Projet Cristal* at `http://pauillac.inria.fr`.

LI, W. AND PINGALI, K. 1994. A singular loop transformation based on non-singular matrices. *International Journal of Parallel Programming 22,* 2 (April).

MASLOV, V. 1995. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *Proc. International Conference on Supercomputing*. 265–269.

MATEEV, N., MENON, V., AND PINGALI, K. 2000. Left-looking to right-looking and vice versa: An application of fractal symbolic analysis to linear algebra code restructuring. In *Proceedings of Euro-Par*. Munich, Germany.

METZGER, R. AND WEN, Z. 2000. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press.

PUGH, W. 1992. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*. 102–114.

RAUCHWERGER, L. AND PADUA, D. A. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems 10,* 2 (February).

REIF, J. H. AND TARJAN, R. E. 1982. Symbolic program analysis in almost linear time. *SIAM Journal on Computing 11,* 1 (February), 81–93.

RINARD, M. C. AND DINIZ, P. C. 1997. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 19,* 6 (November), 942–991.

WEISER, M. 1984. Program slicing. *IEEE Transactions on Software Engineering 10,* 4, 352–357.

WOLF, M. AND LAM, M. 1991. A data locality optimizing algorithm. In *SIGPLAN 1991 conference on Programming Languages Design and Implementation*.

WOLF, M. E., MAYDAN, D. E., AND CHEN, D.-K. 1996. Combining loop transformations considering caches and scheduling. In *MICRO 29*. Silicon Graphics, Mountain View, CA, 274–286.

WOLFE, M. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company.