# Fractal Symbolic Analysis

Nikolay Mateev
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.
mateev@cs.cornell.edu

Vijay Menon
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.
vsm@cs.cornell.edu

Keshav Pingali
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.
pingali@cs.cornell.edu

## ABSTRACT

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, and does not assume any particular interpretation for the operations in that statement. Exploiting the semantics of these operations 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 operations 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 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.

## 1. INTRODUCTION

Modern compilers perform source-level transformations of programs to enhance locality and parallelism. Before a program is transformed, it must be analyzed to ensure that the proposed transformation does not change the input-output behavior of that program. In this paper, we will say that two programs are equal if their input-output behavior is identical (that is, the programs are extensionally equal). Dependence analysis, the most commonly-used analysis technique, computes a partial order between execution instances of statements: a dependence is said to exist from one statement instance to another if one of these instances writes to a memory location that is read or written by the other one [22]. Any reordering of statements consistent with this partial order is permitted since it is guaranteed to leave 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. 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 [3, 8, 21, 13]. These techniques are sufficiently well-understood that they have been incorporated into production compilers such as the SGI MIPSPro.

In spite of these successes, dependence analysis has its shortcomings as is shown by applying statement reordering to the contrived program of Figure 1(a). There are dependences from statement `S1` to `S2`, and from `S1` to `S3`. There are only two statement reorderings consistent with this partial order: the original program, and the program obtained by reordering `S2` and `S3`. In particular, the statement order shown in Figure 1(b) is not consistent with this partial order. However, it is not difficult to verify that the two program fragments in Figure 1 are equivalent. If $x_{in}$ and $y_{in}$ are the values of `x` and `y` at the start, the final value contained in both `x` and `y` are $2 * x_{in}$ for both programs. Therefore, the statement reordering of Figure 1 is legal, even though a compiler that relies on dependence analysis alone will declare that this transformation is illegal. Note that this reordering is legal even if no assumptions are made about algebraic properties of multiplication.

Intuitively, dependence analysis provides *sufficient but not necessary* conditions for the legality of restructuring transformations since this analysis considers only the sets of locations read and written by statements, and does not assume any particular interpretation (meaning) for the operations in the right-hand sides of these statements[1]. For example, the dependences in Figure 1 do not change even if statement `S1` is changed to `y = x*x`, although statement reordering is not legal in the new program. Exploiting the seman-

[0]This work was supported by NSF grants EIA-9726388, ACI-9870687, EIA-9972853, and ACI-0085969.

[1]A more precise variation of dependence analysis is *value-based* dependence analysis [7]. It is easy to verify that this alternative analysis makes no difference in our problem.

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

**Figure 1: Simple Reordering of Statements**

tics of program operations 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.

*Symbolic analysis* is the usual way of exploiting the semantics of operations. To compare two programs for equality, we derive expressions for the outputs of these programs as functions of their inputs, and attempt to prove that these expressions are equal. If an algebraic axiom such as the distributive, associate, or commutative law of arithmetic can be assumed by the compiler, it may be used when proving equality.

In principle, 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 it can also be used to prove equality of programs that implement very different algorithms, such as sorting programs that implement quicksort and mergesort. For example, the effect of the statement reordering shown in Figure 1 can be obtained by using techniques such as value numbering [1] which implicitly rely on symbolic evaluation. However, for all but the simplest programs, symbolic execution and comparison is intractable.

In this paper, we describe *fractal* symbolic analysis which is a novel way of performing symbolic analysis and comparison of a program and its restructured version. Figure 2 illustrates the high-level idea. We assume 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 with only straight-line code, or programs with only straight-line code and 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. On the other hand, if the programs are too complex to be analyzed by the symbolic analyzer, we generate two simplified programs $S_1$ and $T_1$ and try to prove that these simplified programs are themselves equal[2]. The simplified programs have a very special property: *if these simplified programs are equal, we can conclude that the original programs* `S` *and* `T` *are equal as well*. Analysis of the simplified programs is performed in a manner similar to the analysis of the original programs: if the programs are simple enough, they are analyzed by the symbolic analyzer; 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 $S_n$ and $T_n$ that are simple enough to be analyzed even by a symbolic analyzer that can only handle straight-line 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.

There are two important caveats. First, the rules for simplifying programs are derived from the transformation that relates the two

[2] A minor technical detail is that a simplification step may actually produce a number of simplified programs from each of the original programs, in which case it is necessary to establish equality of that number of pairs of simplified programs.

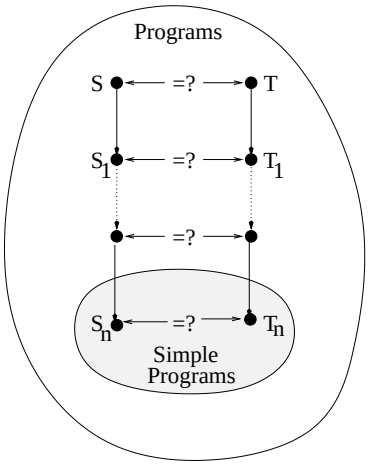



**Figure 2: Overview of Fractal Symbolic Analysis**

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. Second, equality of simplified programs is a sufficient but not in general necessary condition for equality of the original programs. Therefore, successive simplification steps produce programs that are less and less likely to be equal even if the original programs are equal. It is desirable therefore that the core symbolic analyzer be powerful so that recursive simplification can be applied sparingly.

The rest of this paper is organized as follows. In Section 2, we introduce the highlights of our technology by discussing transformation of a small program. In Section 3, we discuss the simplification rules for key transformations in the literature. In Section 4, we describe the base symbolic analyzer we use in our implementation. We apply this technology to automatic blocking of LU factorization with pivoting in Section 5 and show that we can achieve performance comparable with that of the LAPACK library [2] on the SGI Octane. Finally, in Section 6, we discuss ongoing work.

# 2. A SMALL EXAMPLE

In this section, we discuss a small example which is a distillation of LU with partial pivoting and which illustrates various aspects of fractal symbolic analysis.

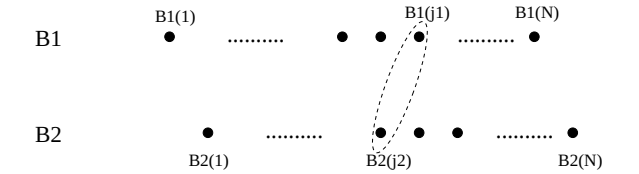
## 2.1 Source and Transformed Programs

The source program of Figure 3(a) traverses an array `A`; at the $j^{th}$ iteration, it swaps elements `A(j)` and `A(j+1)`, and updates all the elements from `A(j+1)` through `A(N)` using the new value in `A(j)`. This 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* will be used to refer to the swap and update statement blocks respectively.

Loop distribution transforms this program into the one shown in Figure 3(b). In this program, all the swaps are done first, and then all the updates are done together. This transformation is useful because the second loop nest is perfectly-nested and can be tiled to get good locality of reference. Are these programs equal?

Dependence analysis requires that there not be a dependence from

```
do j = 1,N-1                    do j = 1,N-1
 B1(j): //swap                    B1(j): //swap
    tmp = A(j);                   tmp = A(j);
    A(j) = A(j+1);                A(j) = A(j+1);
    A(j+1) = tmp;                 A(j+1) = tmp;
 B2(j): //update                do j = 1,N-1
    do i = j+1,N                  B2(j): //update
      A(i) = A(i)/A(j)            do i = j+1,N
                                     A(i) = A(i)/A(j);
 (a) Original Program          (b) Transformed Program
```




(c) Incremental Loop Distribution

```
B1(j1): //swap                  B2(j2)://update
  tmp = A(j1);                    do i = j2+1,N
  A(j1) = A(j1+1);                  A(i) = A(i)/A(j2)
  A(j1+1) = tmp;                B1(j1)://swap
B2(j2): //update                  tmp = A(j1)
  do i = j2+1,N                   A(j1) = A(j1+1);
    A(i) = A(i)/A(j2)             A(j1+1) = tmp;
(d) S1(j1) before S2(j2)      (e) S2(j2) before S1(j1)
```

**Figure 3: Loop Distribution in Running Example**

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. Symbolic analysis of these programs on the other hand is too difficult.

## 2.2 Fractal Symbolic Analysis of Example

The key to fractal symbolic analysis is to consider the transformation of the program in Figure 3(a) to the one in Figure 3(b) not as a one-step transformation but as an incremental process in which instances of *B1* are scheduled before successively earlier instances of *B2* as shown in Figure 3(c). At each step of the incremental process, we locate an instance of block *B2* (say *B2(j2)*) which is executed just *before* an instance of block *B1* (say *B1(j1)* where *j1 > j2*), and reschedule it so that it is executed just *after* that instance. It is obvious that at the end of this process, loop distribution is complete, and that this distribution is legal if each step of the process is legal.

To verify the legality of each step of the incremental process, we must verify that the programs shown in Figures 3(d) and (e) are equal; in other words, that *B1(j1)* and *B2(j2) commute* (assuming that *N > j1 > j2 ≥ 1*). Notice that (i) these programs are simpler than the ones in Figures 3(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 3(d) and (e) 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 3(d). 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 4, consisting of a sequence of guards defining array regions and symbolic expressions for the values in the array in those regions.

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

**Figure 4: Guarded Symbolic Expression for $A_{out}$**

```
B1(j1)://swap                   B2(j2,i)://update body
  tmp = A(j1);                    A(i) = A(i)/A(j2);
  A(j1) = A(j1+1);              B1(j1)://swap
  A(j1+1) = tmp;                  temp = A(j1);
B2(j2,i)://update body            A(j1) = A(j1+1);
  A(i) = A(i)/A(j2);              A(j1+1) = tmp;
(a) B1(j1); B2(j2,i)          (b) B2(j2,i); B1(j1)
```

**Figure 5: Another Step of Simplification**

An identical GSE expresses the result of executing the program of Figure 3(e). If $A$ is assumed to be the only live variable after execution of the two programs, we can conclude that the programs of Figure 3(d) and (e) are equal, so the programs of Figure 3(a) and (b) are also equal.

## 2.3 Discussion

It is useful to understand what happens if we apply another step of simplification to the programs of Figure 3(d) and (e). 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 5(a) and (b); we must verify that *B1(j1)* and *B2(j2,i)* commute, assuming that $N > j1 > j2 \ge 1$ and that $N \ge i > j2$.

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 caveat 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 3(a,b) is not legal.

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

1. What are the simplification rules?
2. What core symbolic analyzer is both powerful and practical?

We answer the first question in Section 3 and the second one in Section 4.

# 3. SIMPLIFICATION RULES

To determine equality of a program and its restructured version, the compiler uses a table similar to the one shown in Table 1(a). To verify legality of a transformation shown in the first column, the Commute function is invoked on appropriate instantiations of program statements as shown in the second column of this table. The intuition behind these legality conditions has been described in Section 2 — the transformation is viewed incrementally as reordering

| Transformation | Legality Condition |
|---|---|
| Statement Reordering<br>`S1; S2;` <-> `S2; S1;` | $Commute(\langle S1, S2\rangle)$ |
| Loop Distribution/Jamming<br>`do i = 1,n`<br>`  S1(i);`<br>`  S2(i);`<br><-><br>`do i = 1,n`<br>`  S1(i);`<br>`do i = 1,n`<br>`  S2(i);` | $Commute(\langle S1(i_1), S2(i_2)\rangle : 1 <= i_2 < i_1 <= n)$ |
| Loop Interchange<br>`do i = 1,n`<br>`  do j = 1,m`<br>`  S(i,j);`<br><-><br>`do j = 1,m`<br>`  do i = 1,n`<br>`  S(i,j);` | $Commute(\langle S(i_1, j_1), S(i_2, j_2)\rangle :$<br>$1 <= i_1 < i_2 <= n \wedge 1 <= j_2 < j_1 <= m)$ |
| Linear Loop Transformation `T`<br>`do (i1,i2,...,ik)`<br>`  S(i1,i2,...,ik);`<br><-><br>`do (i1',i2',...,ik')`<br>`  S'(i1',i2',...,ik')`<br>`where (i1',i2',...,ik') = T(i1,i2,...,ik)` | $Commute(\langle S(\vec{i}), S(\vec{j})\rangle :$<br>$\vec{i} \prec \vec{j} \wedge \mathtt{T}(\vec{i}) \succ \mathtt{T}(\vec{j}))$ |

(a) Simplification Rules for Common Loop Transformations

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

(b) Recursive Simplification Rules

**Table 1: Simplification Rules**

pairs of instances at a time. For lack of space, we have shown the legality conditions for only a few common transformations.

The Commute function is shown in Figure 6. It is passed two program fragments $pgm_1$ and $pgm_2$, some optional *bindings* which are constraints on free variables in the program fragments, and a list of variables that are live at the end of execution of the program fragments. It returns true if $pgm_1$ and $pgm_2$ commute — that is, if the result of executing $pgm_1$ followed by $pgm_2$ is the same as the result of executing them in the opposite order, assuming that we are only interested in the values of live variables at the end of execution.

The Commute function invokes the symbolic analysis and comparison engine directly by calling the Compare method if the program fragments $pgm_1$ and $pgm_2$ are simple enough. Our implementation of this method is discussed in Section 4. On the other hand, if these program fragments are not simple enough, it applies the rules shown in Table 1(b) to simplify the programs recursively till the programs are simple enough to be analyzed symbolically. As with the rules shown in Table 1(a), the rules in Table 1(b) are easy to understand if the restructuring is viewed incrementally. Note that exactly one of the rules in Table 1(b) can be applied at each simplification step, resulting in a deterministic simplification procedure.

## 3.1 Proof of Correctness of Simplification Rules

The validity of the legality conditions in Table 1 follows from the following result, many variations of which have appeared in the literature [11].

THEOREM 1. *Let $S = \{S_1; S_2; S_3; \ldots; S_n\}$ be a sequence of program fragments, and let $p$ be any permutation on $S$. Define $R(p) = \{(S_i, S_j) : 1 \leq i < j \leq n \wedge p(i) > p(j)\}$ as the set of pairs of statements reordered by $p$. Then, the program $S$ is equal to the program $p(S) = \{S_{p(1)}; S_{p(2)}; S_{p(3)}; \ldots; S_{p(n)}\}$ if $\{S_i; S_j\}$ is equal to $\{S_j; S_i\}$ where $(S_i, S_j) \in R(p)$.*

PROOF. The proof follows from induction on the cardinality of $R(p)$. If $\|R(p)\| = 0$ then clearly $S = p(S)$. Otherwise, there must exist an $i$ such that $p(i+1) < p(i)$, which implies that $(S_{p(i+1)}, S_{p(i)}) \in R(p)$. Since $S_{p(i)}$ and $S_{p(i+1)}$ commute and are adjacent in $p(S)$, transposing them gives us an equivalent program $p'(S)$ where $R(p') \subset R(p)$ and $\|R(p')\| = \|R(p)\| - 1$. By induction, $S = p(S)$. □

Intuitively, Theorem 1 allows us to reformulate the problem of checking legality of a transformation as a problem of verifying commutativity of *statement instances* that are reordered by that transformation. The validity of the rules given in Table 1 follows directly from this result.

## 3.2 Reduction Example

An interesting example of the use of the rules in Table 1 is provided by the reduction example shown in Figure 7. The loop interchange shown in Figures 7(a,b) is legal if we assume that addition is commutative and associative, but every iteration of the loop nest reads and writes variable `k`, so dependences are violated by the restructuring. This program is simple enough that symbolic analysis is tractable in principle although it requires reasoning about

```
Commute(pgm1,pgm2,bindings,live_vars) {
  if Simple(pgm1) then
    if Simple(pgm2)
      return Compare({pgm1;pgm2}, {pgm2;pgm1},
        bindings,live_vars)
    else
      return Commute(pgm2,pgm1,bindings,live_vars)
  else //implementation of Table 1(b)
    case (pgm1) {
      //statement composition
      ⟨pgm1';pgm1''⟩ →
        return Commute(pgm1',pgm2,
          bindings,live_vars) ∧ Commute(pgm1'',pgm2,
          bindings,live_vars)

      //conditional
      ⟨ if pred then pgm1' else pgm1''⟩ →
        return Commute(pgm1',pgm2,
          bindings‖pred,live_vars) ∧ Commute(pgm1'',
          pgm2,bindings‖¬pred,live_vars)

      //loop
      ⟨do i = l, u pgm1'(i)⟩ →
        return Commute(pgm1'(i),pgm2,
          bindings‖∀i.l ≤ i ≤ u,live_vars)
    }
}
```

**Figure 6: The *Commute* function**

```
do i = 1,M                  do j = 1,N
  do j = 1,N                  do i = 1,M
  B(i,j)://update on k        B(i,j)://update on k
    k = k + A(i,j)              k = k + A(i,j)
 (a) Original Program        (b) Transformed Program

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

**Figure 7: Loop Interchange of Reduction Code**

summations. In practice most compilers use pattern matching to detect reductions and treat them specially, but pattern matching is notoriously fragile (for example, introducing a temporary into the reduction will cause most pattern matchers to fail).

Fractal symbolic analysis solves the problem elegantly. Using Table 1, we generate the simplified programs shown in Figure 7(c,d). A symbolic analyzer that can analyze straight-line code can deduce easily that these programs equal if it is allowed to assume that addition is commutative and associative; if addition cannot be assumed to have these properties, the symbolic analysis and comparison engine deduces correctly that the program transformation is not legal.

This simple example illustrates an important point. *The framework of fractal symbolic analysis does not require that the objects being computed, such as numbers, obey any algebraic laws.* It is entirely up to the compiler writer to decide whether or not the restructurer is allowed to assume these laws when proving equality of expressions, as described in the next section.

## 3.3 Discussion

As mentioned in Section 2, the precision of fractal symbolic analysis depends on the power of the core symbolic comparison engine. Notice that the procedure in Figure 6 stops simplifying as soon as the statements being compared can be handled by the Compare procedure. A more powerful Compare procedure will result in fewer levels of simplification and potentially more accurate symbolic analysis.

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

**Figure 8: Guarded Symbolic Expressions**

```
Compare(stmt1,stmt2,bindings,live_vars) {
  live1 = set of live altered variables in stmt1
  live2 = set of live altered variables in stmt2
  if(live1 ≠ live2)
    return false

  for each a(k⃗) in live1 {
    tree1 = Build_Expr_Tree(stmt1,a(k⃗),∅)
    tree2 = Build_Expr_Tree(stmt2,a(k⃗),∅)

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

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

**Figure 9: Comparision of Simple Programs**

# 4. SYMBOLIC COMPARISON ENGINE

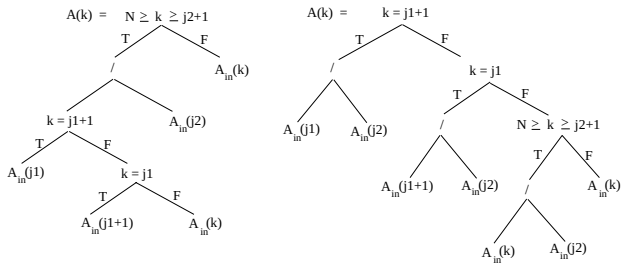
We now describe the core symbolic comparison procedure we use in our implementation.

The symbolic procedure described in this section can compare programs that satisfy the restrictions described below. *It is important to remember that these restrictions are not required of input programs; rather, the* Commute *function of Figure 6 recursively simplifies its program parameters until these conditions are met.* In other words, these conditions must be met by program $S_n$ and $T_n$ in Figure 2, not by programs S and T.

1. Programs consist of assignment statements, for-loops and conditionals. No unstructured control flow is allowed.
2. Loops do not have loop-carried dependences.
3. Array indices and loop bounds are restricted to be affine functions of enclosing loop variables and symbolic constants, and predicates are restricted to be conjunctions and disjunctions of affine inequalities.

The important constraint is the second one. Although a loop may write to a section of an array that is potentially unbounded at compile-time, at most one iteration may affect the value of any given location in an array. This ensures that the symbolic value of a given element of the array can be expressed very simply. We can then summarize the unbounded set of expressions for the values in an entire array with 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 8. Figure 4 shows an example of a GSE.

Figure 9 provides a high-level overview of our symbolic comparison algorithm. We consider each altered scalar or array variable in




(a) Tree for Figure 3(d) (b) Tree for Figure 3(e)

**Figure 10: Conditional Expression Trees for Running Example**

```
Build_Expr_Tree(stmt,tree,bindings) {
  case (tree) {
    Op(op,tree_1,...,tree_n) →
      return Op(op,
          Build_Expr_Tree(stmt,tree_1,bindings),...,
          Build_Expr_Tree(stmt,tree_n,bindings))

    Cond(pred,tree_t,tree_f) →
      return Cond(pred,
          Build_Expr_Tree(stmt,tree_t,bindings),
          Build_Expr_Tree(stmt,tree_f,bindings))

    A(k⃗) →
      case (stmt) {
        ⟨A'(T · i⃗ + c) = tree_1(i⃗)⟩ →
          if (A = A') then
            return Cond(bindings‖k⃗ = T · i⃗ + c,
                tree_1(T^-1 · (k⃗ − c)),A(k⃗))
          else
            return A(k⃗)

        ⟨stmt_1;stmt_2⟩ →
          return Build_Expr_Tree(stmt_1,
            Build_Expr_Tree(stmt_2,tree,bindings),
            bindings)

        ⟨if pred then stmt_1 else stmt_2⟩ →
          return Cond(bindings‖pred,
            Build_Expr_Tree(stmt_1,tree,bindings),
            Build_Expr_Tree(stmt_2,tree,bindings))

        ⟨do i_k = l_k, u_k stmt_1⟩ →
          return Build_Expr_Tree(stmt_1,tree,
            bindings‖∃i_k.l_k ≤ i_k ≤ u_k)
      }
  }
}
```

**Figure 11: Expression Tree Generation**

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

Normalize(tree) {
  case (tree) {
    Op(op,tree_1,...,Cond(pred,tree_t_i,tree_f_i),...,tree_n) →
      return Normalize(Cond(pred,
              Op(op,tree_1,...,tree_t_i,...,tree_n),
              Op(op,tree_1,...,tree_f_i,...,tree_n)))

    Op(op,tree_1,...,tree_n) →
      return Op(op,Normalize(tree_1),...,,Normalize(tree_n))

    Cond(pred,tree_t,tree_f) →
      return Cond(pred,Normalize(tree_t),Normalize(tree_f))

    A(k⃗) →
      return A(k⃗)
  }
}

Flatten(tree,guard) {
  if (guard) then
    case (tree) {
      Cond(pred,tree_t,tree_f) →
        return Flatten(tree_t,guard ∧ pred) ⋃
               Flatten(tree_f,guard ∧ ¬pred)

      Op(op,tree_1,...,tree_n) →
      A(k⃗) →
        return {(guard, expr)}
    }
  else
    return ∅
}
```

**Figure 12: From Expression Trees to GSE's**

the two programs being compared. Note that we only need to consider *live* variables [1]. If the GSE's corresponding to each live altered variable are equal, the two programs are declared to be equal. We now describe how GSE's are constructed and compared.

## 4.1 Generation of Conditional Expression Trees

A guarded symbolic expression is essentially a description of the effect of a program on an array. As an intermediate step towards the construction of this description, we build a symbolic representation of the program that we call a *conditional expression tree*. A conditional expression tree may be viewed as a functional expression that describes the output of the program as a function of its inputs. Figure 10 shows the conditional expression trees for the programs of Figure 3(d,e). 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 straightforward. Figure 11 shows an algorithm to generate such trees. This algorithm 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. Procedure `Build_Expr_Tree` can be described as follows. 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 10(a), this procedure is passed the program of Figure 3(d) as the first parameter, the expression `A(k)` as the second parameter, and the constraint $j1 > j2$ as the third parameter.

## 4.2 Generating Guarded Symbolic Expressions

In general, the conditional expression trees generated above contain a mix of conditions predicated by affine constraints on one hand and arithmetic expressions on the other (Figure 10(a) is an example of such a mixture). To convert these to guarded symbolic expressions, we need to separate the two (Figure 10(b) is an example where the affine constraints are separate from the arithmetic expressions). We accomplish this by converting conditional expression trees into a normal form in which affine constraints are moved above arithmetic expressions by repeated application of the following transformation.

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

  return true
}
```

**Figure 13: Comparision of GSE's**

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

At this point, the guards are generated by flattening the predicates at the top of the normalized expression tree, and the corresponding arithmetic expressions are simply taken from the subtrees beneath these predicates, as is shown in Figure 12.

## 4.3 Comparison of Guarded Symbolic Expressions

Finally, Figure 13 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. 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 [16]. 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 whatever axioms may be assumed to prove equality of expressions. In our current implementation, we just check for *syntactic* equality. This is sufficient both for our simple example and, as we shall see in Section 5, for LU factorization.

## 4.4 Discussion

*It is important to realize that the framework described here does not rely in any way on the algebraic properties of numbers*. If arithmetic operations such as addition can be assumed to be commutative, associative, etc., the corresponding axioms can be used in proving equality of expressions in Figure 13. 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 numbers can be used, and the choice of whether to use these properties or not is under the control of the compiler writer.

# 5. LU WITH PIVOTING

```
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 portion of matrix to right of column j
  do k = j+1, N
    do i = j+1, N
      A(i,k) = A(i,k) - A(i,j)*A(j,k)
```

**Figure 14: LU Factorization 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 14 shows the canonical version of LU factorization with pivoting that appears in the literature [9]. In iteration `j` of the outer loop, computations are performed on column `j` of the matrix `A`, and a portion of the matrix to the right of this column is updated.

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 [2]. These *blocked codes* are too complex to be reproduced here, but they perform much better than the *point version* shown in Figure 14.

   Given point-wise LU factorization with pivoting, can a compiler automatically generate a cache-optimized version by blocking the code? If so, how does the performance of the compiler-optimized code compare with that of hand-blocked code?

2. Right-looking (eager updates) and left-looking (lazy updates) versions of LU factorization can be obtained by interchanging the two loops of the update step. Can a compiler transform right-looking LU to left-looking LU and vice versa?

Fractal symbolic analysis is crucial to address both these challenges. In this paper, we discuss only the problem of blocking; the use of fractal symbolic analysis to convert between left- and right-looking versions is described elsewhere [15].

## 5.1 Automatic Blocking of LU Factorization

To obtain code competitive with LAPACK code, Carr and Lehoucq suggest carrying out the following sequence of restructuring transformations [4].

1. Stripmine the outer loop to formulate 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 out-of-column update.

4. Tile the out-of-column update.

The first two steps, stripmining and index-set-splitting, are trivially legal as they do not reorder any computation. The next step, loop distribution, 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 iteration `B2(m)` to an iteration `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, a compiler that relies on dependence analysis cannot block LU with pivoting using the transformation strategy of Carr and Lehoucq.* Carr and Lehoucq suggest that a compiler may be endowed with application-specific information to recognize the swap and update operations in LU factorization, and to realize that they can be legally interchanged.

We now discuss how our implementation of fractal symbolic analysis, a general-purpose technique, can verify the legality of these transformations. A high level view of the steps of this process is shown in Figure 16. To verify the legality of the loop distribution step, our compiler consults Table 1(a) and determines that it must check if `B1(l)` commutes with `B2(m)` where $jB \leq m < l \leq jB + B - 1$, as shown in Figure 17. However, these simpler programs are not "simple enough"; the loop that computes the pivot in `B1.b(l)` is a recurrence that cannot be handled by our core symbolic comparison engine, as we discussed in Section 4. Therefore, these programs are simplified again using the rule for statement sequences in Table 1(b). This requires the compiler to test whether `B2(m)` commutes with the five subblocks in `B1(l)`. Other than `B1.c(l)`, the subblocks of `B1(l)` touch data that is disjoint from the data touched by `B2(m)`. Therefore, our compiler deduces that these subblocks commute with `B2(m)` (a small detail is that the analysis of whether `B1.b(l)` commutes with `B2(m)` requires an additional step of simplification to eliminate the recurrence in `B1.b(l)`).

The only problem remaining is to demonstrate that `B1.c(l)` and `B2(m)` commute as shown in Figure 18. At this point, these programs are "simple enough", and the Compare method in Figure 9 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 Compare method generates guarded symbolic expressions for `A` from each program. Both GSE's generated from Figure 18 contain six guarded regions, shown pictorially in Figure 19. To prove that the GSE's are actually equivalent, Compare_GSEs is invoked to generate the 36 pairwise intersections, and the Omega library [16] is used to test non-emptiness of these regions. Only six intersections are non-empty (the six regions shown in Figure 19), 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 15. Since the symbolic expressions are syntactically equal, it follows that the restructuring does not change the output of the program even if arithmetic is finite-precision (that is, the transformation is legal even if nothing is assumed about the commutativity and associativity of addition and multiplication).

One important note is that the programs of Figure 18 are equal only if $p(j) \geq j$. Techniques such as value propagation [14, 6] have been developed to perform this type of analysis for indirect array accesses to more accurately compute dependences. 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 1.

With this information, our implementation of fractal symbolic analysis is able to automatically establish the legality of the loop distribution transformation in Figure 15. For this example, our implementation, prototyped in Caml-Light [12], 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.

## 5.2 Experimental Results

Figure 20 shows the improvement in performance that results from blocking LU with pivoting as discussed above. The lowest line (labeled `Right-looking LU`) shows that the SGI compiler is not able to block the original code shown in Figure 14. However, if the loop distribution of Figure 15 is performed by hand, the SGI compiler can effectively block the resulting code. The compiler is able to automatically tile the right-looking update (`B2`) and essentially accomplish the last Carr/Lehoucq step listed above. Once we have isolated the update, standard dependence analysis will allow us to distribute the portion of the swap computation to left and right of the current block column outside the block column computation to obtain a blocking structure almost identical to that in the Netlib LAPACK. The SGI compiler does not do this itself; when we apply this transformation by hand, we see a modest increase in performance. The resulting performance is shown by the line labeled `Distributed update`.

Nevertheless, this code, at 200 MFlops, is still a factor of two slower than the LAPACK codes. Further experimentation found the remaining performance gap due the compiler's suboptimal treatment of the right-looking update computation. 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, we note that if we replace 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.

We conclude that a compiler which uses fractal symbolic analysis should be able to restructure LU with pivoting and obtain performance comparable to that of the LAPACK library code, provided the performance of compiler-generated BLAS improves.

# 6. RELATED AND FUTURE WORK

A simple kind of symbolic analysis called *value numbering* [1] and a generalization called *global value numbering* [18] 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 [10] and by Rauchwerger and Padua [17], but their

```
  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-1, N
        do i = j+1, N
          A(i,k) = A(i,k) - A(i,j)*A(j,k)
```

(b) After Loop Distribution

**Figure 15: LU Factorization: Distribution Step**

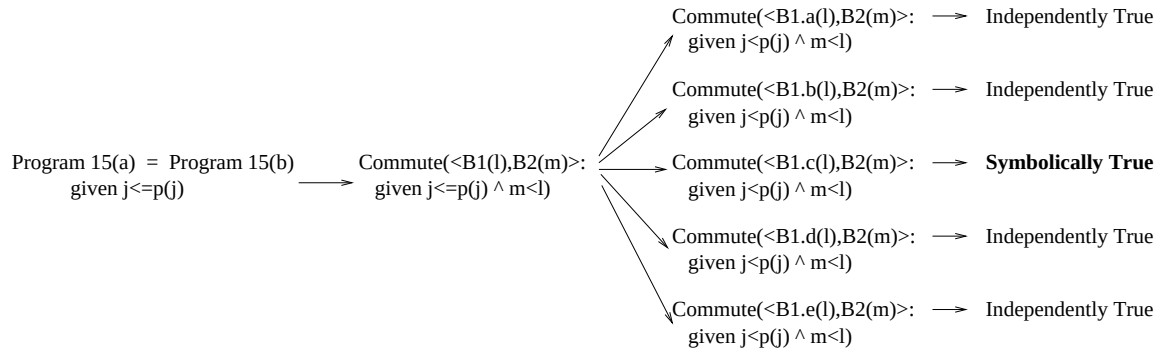



**Figure 16: Commute Calls for LU**

```
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)
```

(a) `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)
```

(b) `B2(m); B1(l)`

**Figure 17: Simplified Comparison #1**

```
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)
```

(a) `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
```

(b) `B2(m); B1.c(l)`

**Figure 18: Simplified Comparison #2**

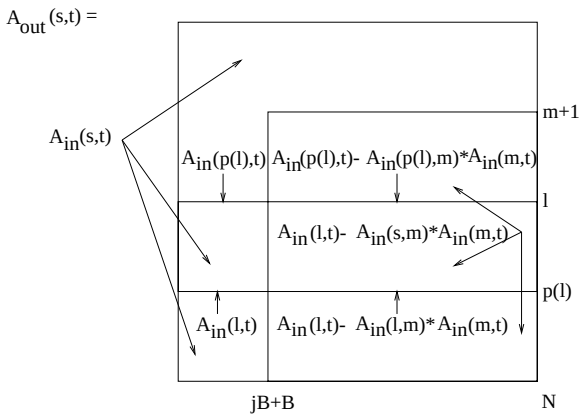



**Figure 19: Regions and Expressions for Simplified LU**

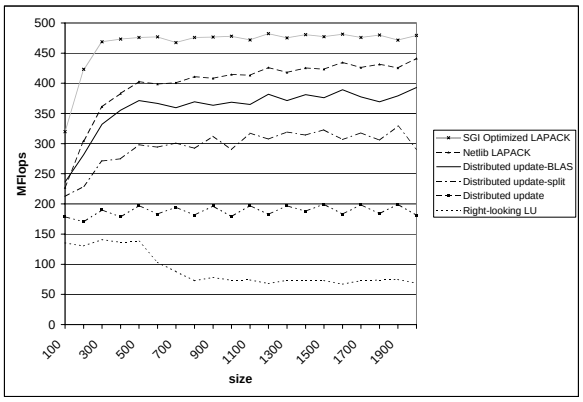



**Figure 20: Experimental Results**

goal is to perform strength increasing to eliminate loop-carried dependences for automatic parallelization. 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 since this may eliminate the need for recursive simplification in such programs.

The work that is closest to our own is Rinard's *commutativity analysis* [19] which 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. We are interested in proving the correctness of program transformations, not in parallelizing programs, and requiring all operations to commute with each other is too strong a condition for our application; there is also no analog of recursive simplification in commutativity analysis.

The algorithm for generating guarded symbolic expressions in Section 4 is reminiscent of backward slicing [20] 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 work described in this paper can be extended in many ways such as the following.

- 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 [5] to compare such expressions using the usual algebraic axioms of numbers. 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 computer arithmetic does not necessarily obey these algebraic axioms, so these axioms must be used with care.

- The intuition behind fractal symbolic analysis is to view a program transformation as a process which transforms the initial program incrementally to the final program. In general, there are many incremental processes that achieve the effect of a given transformation, and Table 1 shows just one such way for the transformations listed there.

  Given a program and its transformation, fractal symbolic analysis may succeed in proving the correctness of some of these processes, but it may fail conservatively for others. A useful analogy is induction which can be viewed as an incremental

way of proving predicates: given a predicate, we can usually formulate many inductive strategies for proving it, but only some of them will actually succeed.

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 5 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* transformation sequences [3, 21, 13]. 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.

  One possibility is to compute dependence information, and then eliminate some of the apparent dependences by performing fractal symbolic analysis. For this strategy to work well, it is necessary to perform analysis at the right level of granularity; dependences between statement instances 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. For example, in LU with pivoting, the pivot block and the update block must considered in their entirety to establish the legality of the loop distribution discussed earlier.

  How does a compiler determine the right level of granularity at which it should do fractal symbolic analysis?

We leave these questions for future work.

# 7. REFERENCES


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, MA, second edition, 1986.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition*. SIAM, Philadelphia, 1995.

[3] U. Banerjee. A theory of loop permutations. In *Languages and compilers for parallel computing*, pages 54–74, 1989.

[4] S. Carr and R. B. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Transactions on Mathematical Software*, 23(3):336–361, September 1997.

[5] B. Char, K. Geddes, and G. Gonnet. The Maple symbolic computation system. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 17(3/4):31–42, August/November 1983.

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

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

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

[9] G. Golub and C. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[10] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

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

[12] X. Leroy. The Caml Light system, release 0.71. Documentation and user's manual. Technical report, INRIA, March 1996. Available from *Projet Cristal* at `http://pauillac.inria.fr`.

[13] W. Li and K. Pingali. A singular loop transformation based on non-singular matrices. *International Journal of Parallel Programming*, 22(2), April 1994.

[14] V. Maslov. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *Proc. International Conference on Supercomputing*, pages 265–269, July 1995.

[15] N. Mateev, V. Menon, and K. Pingali. 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, August 29 – September 1, 2000.

[16] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, pages 102–114, Aug. 1992.

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

[18] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal on Computing*, 11(1):81–93, February 1982.

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

[20] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[21] M. Wolf and M. Lam. A data locality optimizing algorithm. In *SIGPLAN 1991 conference on Programming Languages Design and Implementation*, June 1991.

[22] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.