Fractal Symbolic Analysis

Keshav Pingali

Joint work with
Vijay Menon, Nikolai Mateev
Cornell University

Context

- Restructuring compilers
- Program Transformations
  - Legality of Transformation
    - Must preserve semantics of original program
  - Generation of Transformation
    - Enhance temporal/spatial locality
    - Increase parallelism
- Focus of talk: legality

Legality of transformations

- Standard approach: dependence analysis
  - Sufficient but not necessary condition for legality
  - Not powerful enough to handle LU + pivoting
- Powerful approach: symbolic analysis
  - Intractable for most programs
- Our approach: fractal symbolic analysis
  - Combines power with tractability
  - Solves problem with restructuring LU + pivoting!

Overview of Talk

- Background on Legality
  - Dependence Analysis
  - Symbolic Execution
  - Two running examples
- Fractal Symbolic Analysis
  - Automatic Blocking of LU with pivoting
- Summary and Open Issues
**Dependence Analysis**
- Considers only memory locations touched by statements
  - Independent operations may be reordered
  
  \[
  \begin{align*}
  S1; & \quad \longleftrightarrow \quad S2; \\
  S2 & \quad \longrightarrow \quad S1
  \end{align*}
  \]
  - S1 does not read/write location written by S2 and vice versa
- Extends to loop transformations
  - interchange
  - distribution
  - tiling

**Symbolic Analysis**
- Dependence Analysis is inexact:
  \[
  \begin{align*}
  s1: & \quad a = b \\
  s2: & \quad b = 2*b \\
  s3: & \quad a = 2*a
  \end{align*}
  \]
- Symbolic execution shows equality:
  \[
  \begin{align*}
  a_{out} & = 2*b_{in} \\
  b_{out} & = 2*b_{in}
  \end{align*}
  \]
  - Powerful and general technique
  - But, intractable for recurrent loops....

**Example #1: Reduction**
- Loop interchange for spatial locality

\[
\begin{align*}
\text{do } i & = 1,N \\
\text{do } j & = 1,N \\
k & = k + A(i,j)
\end{align*}
\]

- Pattern matching is fragile. Consider:

\[
\begin{align*}
\text{do } i & = 1,N \\
\text{do } j & = 1,N \\
t1 & = k \\
t2 & = t1 + A(i,j) \\
k & = t2
\end{align*}
\]

- SGI MIPSpro fails to interchange above
- Symbolic execution?
  - body is equivalent to previous example
  - how to handle recurrent loop?
Example #2: Pivoting

- Loop distribution (assume $p(j) \geq j$)

\[
\text{for } j = 1:n \\
\text{tmp} = a(j) \\
\text{a(j)} = a(p(j)) \\
\text{a(p(j))} = \text{tmp}
\]

\[\text{B1(j):} \]

\[
\text{for } j = 1:n \\
\text{tmp} = a(j) \\
\text{a(j)} = a(p(j)) \\
\text{a(p(j))} = \text{tmp}
\]

\[\text{B2(j):} \]

\[
\text{for } i = j+1:n \\
\text{a(i)} = a(i)/a(j)
\]

- Dependence analysis: too conservative
- Symbolic comparison: ???

Pivoting, Cont.

- Distribution reorders
  - swaps
  - updates

\[\text{Swap}(j) \xrightarrow{\text{Update}(j)} \]

- Effect of distribution
  - Before: swaps & updates interleaved
  - After: all swaps followed by all updates

Overview of Talk

- Background on Legality
- Fractal Symbolic Analysis
  - High Level Algorithm
  - Examples
  - Guarded Symbolic Expressions
- Automatic Blocking of LU with pivoting
- Summary and Open Issues

High Level Algorithm

- Compare a program $P_1$ and its transformed version $P_2$ via comparison of simplified programs

\[
\begin{align*}
P_1 & =?= P_2 \\
\downarrow & \uparrow \downarrow \\
P_1' & =?= P_2' \\
\downarrow & \uparrow \downarrow \\
P_1'' & =?= P_2''
\end{align*}
\]

- Equality of simpler programs $\Rightarrow$ equality of complex programs
  - sufficient, but not necessary condition
  - Simplify until symbolic execution is tractable
  - e.g., comparing basic blocks
Example

- Prove or disprove equivalence of:

  \[
  \text{for } i = 1 : n \\
  S1(i); \\
  S2(i); \\
  \]

  vs.

  \[
  \text{for } i = 1 : n \\
  S1(i); \\
  S2(i); \\
  \]

Inductive Approach

- Think of transformation as incremental process

  \[
  S2(1) \quad S2(2) \quad S2(3) \quad \ldots \quad S2(n-1) \quad S2(n) \\
  S1(1) \quad S1(2) \quad S1(3) \quad \ldots \quad S1(n) \\
  \]

- If reordering at each step is legal, overall transformation is legal!

Theorem

1. Any permutation can be generated by sequences of adjacent transpositions.
2. The reordered pairs of a permutation generate such a sequence of adjacent transpositions.

(1,4,2,5,3,6) \rightarrow (1,2,3,4,5,6) reordered pairs: (4,2),(4,3),(5,3)

Incremental proof of legality

- Loop distribution: show that

  \[
  \forall l,m. (1 \leq m < l \leq n) \\
  S2(m) \land S1(l) = S1(l) \land S2(m) \\
  \]

- Similar conditions:
  - Statement reordering
  - Loop interchange
  - Loop reversal
  - Loop tiling

- Proving incremental steps may be easier than proving that entire transformation is correct.
Example #1: Loop Interchange

- Prove equivalence of:

  \[
  \begin{align*}
  &\text{do } i = 1, N \\
  &\text{do } j = 1, N \\
  &t_1 = k \\
  &t_2 = t_1 + A(i,j) \\
  &k = t_2
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{do } j = 1, N \\
  &\text{do } i = 1, N \\
  &t_1 = k \\
  &t_2 = t_1 + A(i,j) \\
  &k = t_2
  \end{align*}
  \]

- Sufficient condition:

  - Prove
    - \( S(i_1,j_1); S(i_2,j_2) = S(i_2,j_2); S(i_1,j_1) \)
    - where \( i_1 < i_2 \) and \( j_1 > j_2 \)

Simplified Test

- Prove equivalence:

  \[
  \begin{align*}
  &t_1 = k \\
  &t_2 = t_1 + A(i,j) \\
  &k = t_2 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  &t_1 = k \\
  &t_2 = t_1 + A(i,j) \\
  &k = t_2 \\
  \end{align*}
  \]

- Proves legality of interchange

  - Sufficient, but not necessary condition

Example #2: Loop Distribution

- Given \( p(j) \geq j \), prove:

  \[
  \begin{align*}
  &\text{for } j = 1:n \\
  &\text{tmp } = a(j) \\
  &B1(j): \ a(j) = a(p(j)) \quad a(p(j)) = \text{tmp} \\
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{for } i = j+1:n \\
  &a(i) = a(j)/a(j) \\
  &B2(j): \ a(i) = a(i)/a(j) \\
  \end{align*}
  \]

- Dependence analysis: too conservative
- Symbolic comparison: intractable

Simplified Test

- Given \( p(l) \geq l \land l > m \) prove:

  \[
  \begin{align*}
  &\text{for } i = m+1:n \\
  &a(i) = a(i)/a(m) \\
  &B1(l): \ a(l) = a(p(l)) \quad a(p(l)) = \text{tmp} \\
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{for } i = m+1:n \\
  &a(i) = a(i)/a(m) \\
  &B2(m): \ a(i) = a(i)/a(m) \\
  \end{align*}
  \]

- Further simplification?
Another Step

- Given $p(l) \geq l \land l > m \land i > m$ prove:

  
  \[
  \begin{align*}
  a(i) &= a(i)/a(m) & \text{tmp} &= a(l) \\
  \text{tmp} &= a(l) & a(l) &= a(p(l)) \\
  a(l) &= a(p(l)) & \text{tmp} &= \text{tmp} \\
  a(p(l)) &= \text{tmp} & a(i) &= a(i)/a(m)
  \end{align*}
  \]

  - Programs not equivalent
  - Over-simplification!

Observation

- Underlying symbolic technique is important
- More powerful symbolic analysis
  - less simplification
  - more accurate test

Our Symbolic Analyzer

- Restriction
  - non-recurrent loops (no dependences)
  - affine indices/loop bounds (finite array regions)

- Restricted Programs
  - Easily identified
  - Can symbolically summarize effect

Conditional Symbolic Expressions

- Summarize effect of statement on data
- For single statement:

  
  \[
  a(p(l)) = \text{tmp}
  \]

  \[
  a_{\text{out}}(k) = \begin{cases} 
  (k = p(l)) \Rightarrow \text{tmp} \\
  \text{else} \Rightarrow a_{i}(k)
  \end{cases}
  \]

  - For multiple statements, recurse
Back to Example #2

- 1 level of simplification meets restriction:

\[ \text{B2(m): } \begin{align*} 
    & \text{for } i = m+1:n \\
    & a(i) = a(i)/a(m) \\
    \end{align*} \]

\[ \text{B1(l): } \begin{align*} 
    & \text{tmp = } a(l) \\
    & a(l) = a(p(l)) \\
    \end{align*} \]

where \( p(l) \geq 1 \land l > m \)

- No further simplification necessary

Conditional Expression Tree #1

Expression live output variables in terms of input variables

Conditional Expression Tree #2

Normalization

Rotate conditions to top

Symbolic Program Transformation for Numerical Codes
Guarded Symbolic Expressions

- Convert to GSE:
  - \( \text{array}_{out}(j) = \begin{cases} \text{guard}_i(j) \Rightarrow \text{expr}_i(j) \\ \text{guard}_n(j) \Rightarrow \text{expr}_n(j) \end{cases} \)

- guards
  - affine constraints on
    - loop vars
    - symbolic constants
  - describe regions of array

- exprs
  - unconditional symbolic expressions
  - describe values in an array region

Comparing GSE’s

- Comparison of GSE’s:
  - Prove union of guards in each GSE are equivalent
    - GSE’s must cover same regions
  - When guards of 2 GSE's intersect,
    - Prove corresponding expressions equivalent

- Tools:
  - Integer Programming (e.g., Omega Library)
  - Symbolic Math Engine (e.g., Maple)

Back to Example

- For both program blocks:
  - \( a_{out}(k) = \begin{cases} k \leq m & \Rightarrow a_y(k) \\ k = l & \Rightarrow a_y(p(l))/a_y(m) \\ k = p(l) & \Rightarrow a_y(l)/a_y(m) \\ \text{else} & \Rightarrow a_y(k)/a_y(m) \end{cases} \)

- Note:
  - 16 pair wise intersections / 4 non-empty
  - Expressions are syntactically identical
  - No floating point computation reordered!

Loop distribution is legal in our example

```
for j = 1:n
  tmp = a(j)
  B1(j): a(j) = a(p(j))
  a(p(j)) = tmp
  B1(j): a(j) = a(p(j))

B2(j): for i = j+1:n
  a(i) = a(i)/a(j)

for j = 1:n
  tmp = a(j)
  B1(j): a(j) = a(p(j))
  a(p(j)) = tmp
  B1(j): a(j) = a(p(j))

B2(j): for i = j+1:n
  a(i) = a(i)/a(j)
```
Overview of Talk

- Background on Legality
- Fractal Symbolic Analysis
- Automatic Blocking of LU with pivoting
  - Blocking LU
  - Legality Issue
  - Application of FSA
- Summary and Open Issues

LU Factorization with Partial Pivoting

- Key algorithm for solving systems of linear equations:
  - To solve $Ax = b$ for $x$:
    - => Factor $A$ into $LU$
      - $L$ is lower triangular
      - $U$ is upper triangular
    - => Solve $Ly = b$ for $y$
      - Forward substitution
    - => Solve $Ux = y$ for $x$
      - Backward substitution
  - Note:
    - Partial pivoting required for stability
    - Data cache key to performance

Blocking LU without Pivoting

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

- Must be blocked to exploit reuse in update
- Compiler transformations (Carr & Kennedy 1992)
  - strip-mining
  - index-set-splitting
  - loop distribution
  - tiling

LU with Partial Pivoting

```
do j = 1, N
  p(j) = j;
  do i = j+1, N
    if (A(i,j) > A(p(j),j))
      p(j) = i;
  do k = 1, N
    tmp = A(j,k);
    A(j,k) = A(p(j),k);
    A(p(j),k) = tmp;
```

- Same opts. ⇒ legal blocked code
Caveat: Proving Legality

- Blocking is legal, but
- Reorders swaps and updates
- Violates dependences

Legality: Loop Distribution

```
do jB = 1,N,B
  do j = jB,jB+B-1
    p(j) = j;
  do i = j+1,N
    if (A(i,j)>A(p(j),j))
      p(j) = i;
  do k = 1,N
    tmp = A(j,k);
    do jB = 1,N,B
      do j = jB,jB+B-1
        p(j) = j;
      do i = j+1,N
        if (A(i,j)>A(p(j),j))
          p(j) = i;
      do k = 1,N
        tmp = A(j,k);
        do jB = 1,N,B
          do j = jB,jB+B-1
            p(j) = j;
          do i = j+1,N
            if (A(i,j)>A(p(j),j))
              p(j) = i;
          do k = 1,N
            tmp = A(j,k);
            do j = jB,jB+B-1
              do k = jB+B,N
                do i = j+1,N
                  A(i,k) = A(i,k) - A(i,j)*A(j,k);
                do k = jB+B,N
                  do i = j+1,N
                    A(i,k) = A(i,k) - A(i,j)*A(j,k);
              do jB = 1,N,B
                do j = jB,jB+B-1
                  do k = jB+B,N
                    do i = j+1,N
                      A(i,k) = A(i,k) - A(i,j)*A(j,k);
```

Simplified Programs

```
B1(l):
  pl = l;
  do i = l+1,N
    if (A(i,l)>A(pl,l))
      pl = i;
  do k = 1,N
    tmp = A(l,k);
    A(l,k) = A(pl,k);
    A(pl,k) = tmp;
  do i = l+1,N
    A(i,l) = A(i,l)/A(l,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,k) - A(i,m)*A(m,k);
```

Another step of simplification

```
B3(l):
  do k = 1,N
    tmp = A(l,k);
    A(l,k) = A(pl,k);
    A(pl,k) = tmp;
  do i = l+1,N
    A(i,l) = A(i,l)/A(l,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);
```

• Prove equivalence where \( jB \leq m < l \leq pl, jB+B-1 \)
Nonempty Intersecting Regions

Let's consider the following conditions:

1. \( A(l,y) - A(l,m) \cdot A(m,y) = A(l,y) - A(l,m) \cdot A(m,y) \)
   \[ \{[l,y] \mid 1 \leq j_B \leq m < l \leq j_E < y \leq N \land l \leq p_l \leq N \} \]

2. \( A(pl,y) - A(pl,m) \cdot A(m,y) = A(pl,y) - A(pl,m) \cdot A(m,y) \)
   \[ \{[l,y] \mid 1 \leq j_B \leq m < l < pl \leq N \land l, y \leq j_E \leq N \land 1 \leq y \} \]

3. \( A(x,y) - A(x,m) \cdot A(m,y) = A(x,y) - A(x,m) \cdot A(m,y) \)
   \[ \{[x,y] \mid 1 \leq j_B \leq m < x < l \leq j_E < y \leq N \land l \leq pl \leq N \} \cup \{[x,y] \mid 1 \leq j_B \leq m < l \leq j_E < y \leq N \land l < x < pl \leq N \} \]

Note: no floating point computation reordered!

Conditional Tree Expressions

Region("[x,y]: 1 \leq j_B \leq m \leq l \leq pl \leq N \land l \leq j_E \leq N \land 1 \leq x, y \leq N",
Cond("x=pl \land (exists [k2]:1\leq k2\leq N \land y=k2)",
Cond("(exists [k4,i4]:j_E+1\leq k4\leq N \land m+1\leq i4\leq N \land l=i4 \land y=k4)",
Op("-",
Leaf("A(l,y)"),
Op("*",
Leaf("A(l,m)"),
Leaf("A(m,y)")),
Leaf("A(l,y)")),
Cond("x=l \land (exists [k2]:1\leq k2\leq N \land y=k2)",
Cond("(exists [k4,i4]:j_E+1\leq k4\leq N \land m+1\leq i4\leq N \land pl=i4 \land y=k4)",
Op("-",
Leaf("A(pl,y)"),
Op("*",
Leaf("A(pl,m)"),
Leaf("A(m,y)")),
Leaf("A(pl,y)")),
Cond("(exists [k4,i4]:j_E+1\leq k4\leq N \land m+1\leq i4\leq N \land x=i4 \land y=k4)",
Op("-",
Leaf("A(x,y)"),
Op("*",
Leaf("A(x,m)"),
Leaf("A(m,y)")),
Leaf("A(x,y)"))));

Six Regions

- \( A_{out}(i,j) = \)

\[
\begin{align*}
A_{in}(i,j) & \quad \text{(original)} \\
A_{in}(p(l),j) & \quad \text{(partitioned)} \\
A_{in}(l,j) & \quad \text{(partitioned)} \\
A_{in}(p(l),j) - A_{in}(p(l),j) \cdot A_{in}(m,j) & \quad \text{(partitioned)} \\
A_{in}(l,j) - A_{in}(l,j) \cdot A_{in}(m,j) & \quad \text{(partitioned)} \\
A_{in}(i,j) - A_{in}(i,j) \cdot A_{in}(m,j) & \quad \text{(partitioned)} \\
\end{align*}
\]

LU Performance

![LU Performance Graph](image)

- 300 MHz SGI Octane with 2MB L2 Cache
Summary of Fractal Symbolic Analysis

- Tractable approach to using symbolic analysis to prove legality of program transformations
- Enables tradeoff between
  - tractability (dependence analysis)
  - accuracy (symbolic comparison)
- Encapsulates symbolic information a compiler is permitted to use
- Prototype implemented in OCAML
- Solves problem of restructuring LU + pivoting

Related Work

- Haghighat and Polychronopoulos (1996)
  - Symbolic analysis for induction variable recognition

- Fahringer and Scholz (1997)
  - Symbolic dependence testing

- Rinard (1997)
  - Commutativity analysis for parallelization

Open Issues

- Synthesis of Transformations
  - e.g., dependence vectors ⇒ transformations

- Better underlying symbolic analysis
- Performance: how do we apply this to large programs?