Using Learning Techniques in Invariant Inference

Alex Aiken
Aditya Nori
Rahul Sharma
Saurabh Gupta
Bharath Hariharan
Invariant Inference

• An old problem

• A different approach with two ideas:
  1. Separate invariant inference from the rest of the verification problem
Why?

for (B) {
  ... code ...
}

Pre

Pre )I

I \AE B
{ code }
I

Post

I \AE: B )
Post
Invariant Inference

• An old problem

• A different approach with two ideas:
  1. Separate invariant inference from the rest of the verification problem
  2. Guess the invariant from executions
Why?

• Complementary to static analysis
  - underapproximations
  - “see through” hard analysis problems
    • functionality may be simpler than the code

• Possible to generate many, many tests
Nothing New Under the Sun

• Sounds like DAIKON?
  - Yes!

• Hypothesize (many) invariants
  - Run the program
  - Discard candidate invariants that are falsified
  - Attempt to verify the remaining candidates
A Simple Program

s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + 1;
    y := y + 1;
}

• Instrument loop head

• Collect state of program variables on each iteration
A DAIKON-Like Approach

Hypothesize
- $s = y$
- $s = 2y$

Data

<table>
<thead>
<tr>
<th>$s$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
A DAIKON-Like Approach

\[ s = 0; \]
\[ y = 0; \]
\[ \text{while}(\ast)\{ \]
\[ \text{print}(s,y); \]
\[ s := s + 1; \]
\[ y := y + 1; \]
\[ \}\]

- Hypothesize
  - \( s = y \)
  - \( s = 2y \)

- Data

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
A DAIKON-Like Approach

\[ s = 0; \]
\[ y = 0; \]
while( * )
{
    print(s,y);
    s := s + 1;
    y := y + 1;
}

- Hypothesize
  - \( s = y \)
  - \( s = 2y \)

- Data

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Another Approach

\( s = 0; \)
\( y = 0; \)
\textbf{while}( * )
{
 \textbf{print}(s,y); \\
 s := s + 1; \\
 y := y + 1; \\
}

\begin{tabular}{|c|c|}
\hline
s   & y  \\
\hline
0   & 0  \\
1   & 1  \\
2   & 2  \\
3   & 3  \\
\hline
\end{tabular}
Arbitrary Linear Invariant

\[ as + by = 0 \]

• Data

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Observation

\[as + by = 0\]

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[w = a = 0, b = 0\]
Observation

\[ as + by = 0 \]

\[ \{ w \mid Mw = 0 \} \]

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{w} & = 0 \\
\text{a} & = 0 \\
\text{b} & = 0
\end{align*}
\]
Observation

$$as + by = 0$$

$$\text{NullSpace}(M)$$

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

$$w = a = b = 0$$
Linear Invariants

• Construct matrix $M$ of observations of all program variables

• Compute $\text{NullSpace}(M)$

• All invariants are in the null space
Spurious “Invariants”

• All invariants are in the null space
  - But not all vectors in the null space are invariants

• Consider the matrix
  \[
  \begin{pmatrix}
  s & y \\
  0 & 0
  \end{pmatrix}
  \]

• Need a check phase
  - Verify the candidate is in fact an invariant
An Algorithm

• Check candidate invariant
  - If an invariant, done
  
  - If not an invariant, get counterexample
    • A reachable assignment of program variables falsifying the candidate

• Add new row to matrix
  - And repeat
Termination

- How many times can the solve & verify loop repeat?

- Each counterexample is linearly independent of previous entries in the matrix

- So at most $N$ iterations
  - Where $N$ is the number of columns
  - Upper bound on steps to reach a full rank matrix
Summary

• Superset of all linear invariants can be obtained by a standard matrix calculation

• Counter-example driven improvements to eliminate all but the true invariants
  - Guaranteed to terminate
What About Non-Linear Invariants?

s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + y;
    y := y + 1;
}

Idea

• Collect data as before

• But add more columns to the matrix
  - For derived quantities
  - For example, $y^2$ and $s^2$

• How to limit the number of columns?
  - All monomials up to a chosen degree $d$

[Nguyen, Kapur, Weimer, Forrest 2012]
What About Non-Linear Invariants?

\[ s = 0; \]
\[ y = 0; \]
\[ \text{while} (\ * \ ) \{ \]
\[ \text{print}(s,y); \]
\[ s := s + y; \]
\[ y := y + 1; \]
\[ \} \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>s</th>
<th>y</th>
<th>s^2</th>
<th>y^2</th>
<th>sy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td>36</td>
<td>9</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>4</td>
<td>100</td>
<td>16</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
Solve for the Null Space

\[ a + bs + cy + ds^2 + ey^2 + fsy = 0 \]

<table>
<thead>
<tr>
<th>1</th>
<th>s</th>
<th>y</th>
<th>s^2</th>
<th>y^2</th>
<th>sy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td>36</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>4</td>
<td>100</td>
<td>16</td>
<td>40</td>
</tr>
</tbody>
</table>

Candidate invariant: \[-2s + y + y^2 = 0\]
Comments

• Same issues as before
  - Must check candidate is implied by precondition, is inductive, and implies the postcondition on termination

  - Termination of invariant inference guaranteed if the verifier can generate counterexamples

• Experience: Solvers do well as checkers!
## Experiments

<table>
<thead>
<tr>
<th>Name</th>
<th>#vars</th>
<th>deg</th>
<th>Data</th>
<th>#and</th>
<th>Guess time (sec)</th>
<th>Check time (sec)</th>
<th>Total time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul2</td>
<td>4</td>
<td>2</td>
<td>75</td>
<td>1</td>
<td>0.0007</td>
<td>0.010</td>
<td>0.0107</td>
</tr>
<tr>
<td>LCM/GCD</td>
<td>6</td>
<td>2</td>
<td>329</td>
<td>1</td>
<td>0.004</td>
<td>0.012</td>
<td>0.016</td>
</tr>
<tr>
<td>Div</td>
<td>6</td>
<td>2</td>
<td>343</td>
<td>3</td>
<td>0.454</td>
<td>0.134</td>
<td>0.588</td>
</tr>
<tr>
<td>Bezout</td>
<td>8</td>
<td>2</td>
<td>362</td>
<td>5</td>
<td>0.765</td>
<td>0.149</td>
<td>0.914</td>
</tr>
<tr>
<td>Factor</td>
<td>5</td>
<td>3</td>
<td>100</td>
<td>1</td>
<td>0.002</td>
<td>0.010</td>
<td>0.012</td>
</tr>
<tr>
<td>Prod</td>
<td>5</td>
<td>2</td>
<td>84</td>
<td>1</td>
<td>0.0007</td>
<td>0.011</td>
<td>0.0117</td>
</tr>
<tr>
<td>Petter</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>1</td>
<td>0.0003</td>
<td>0.012</td>
<td>0.0123</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>6</td>
<td>2</td>
<td>362</td>
<td>1</td>
<td>0.003</td>
<td>0.015</td>
<td>0.018</td>
</tr>
<tr>
<td>Cubes</td>
<td>4</td>
<td>3</td>
<td>31</td>
<td>10</td>
<td>0.014</td>
<td>0.062</td>
<td>0.076</td>
</tr>
<tr>
<td>geoReihe1</td>
<td>3</td>
<td>2</td>
<td>25</td>
<td>1</td>
<td>0.0003</td>
<td>0.010</td>
<td>0.0103</td>
</tr>
<tr>
<td>geoReihe2</td>
<td>3</td>
<td>2</td>
<td>25</td>
<td>1</td>
<td>0.0004</td>
<td>0.017</td>
<td>0.0174</td>
</tr>
<tr>
<td>geoReihe3</td>
<td>4</td>
<td>3</td>
<td>125</td>
<td>1</td>
<td>0.001</td>
<td>0.010</td>
<td>0.011</td>
</tr>
<tr>
<td>potSumm1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0.0002</td>
<td>0.011</td>
<td>0.0112</td>
</tr>
<tr>
<td>potSumm2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0.0002</td>
<td>0.009</td>
<td>0.0092</td>
</tr>
<tr>
<td>potSumm3</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>0.0002</td>
<td>0.012</td>
<td>0.0122</td>
</tr>
<tr>
<td>potSumm4</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>1</td>
<td>0.0002</td>
<td>0.010</td>
<td>0.0102</td>
</tr>
</tbody>
</table>
Summary to This Point

• Sound and complete algorithm for algebraic invariants
  - Up to a given degree

• Guess and Check
  - Hard part is inference done by matrix solve
  - Check part done by standard SMT solver
  - Much simpler and faster than previous approaches
What About Disjunctive Invariants?

- Disjunctions are expensive

- Existing techniques severely restrict disjunctions
  - E.g., to a template
Good States
Separating Good States and Bad States
Separating Good States and Bad States
More Precisely . . .

- **A state** is a valuation of program variables

- **Correct programs have good and bad states**
  - All reachable states are good
    - Because we assume the program is correct
  - **Assertions define the bad states**
    - States that would result in the assertion being violated

- **An invariant is a separator**
  - Of the good states from the bad states
From Verification to Machine Learning

• From data we want to learn a separator of the good and bad states

• This is a machine learning problem
Goals

• Produce boolean combination of linear inequalities
  - Without templates

• Predictive
  - Generalizes well from small test suite

• Efficient
  - Hard, but more on this later
PAC Learning

• Given some positive and negative examples
  - Learn separator

• Separator is Probably Approximately Correct
  - With confidence $1 - x$ the accuracy is $1 - e$
  - The number of examples is $m = \text{poly}(1/x, 1/e, d)$
Example for Good and Bad States

\begin{align*}
x &:= y; \\
\text{while} (x \neq 0) \text{ do} \\
&\quad x := x - 1; \\
&\quad y := y - 1; \\
\text{assert } y = 0
\end{align*}

- **Good states:**
  - \((x, y) = (1, 1), (2, 2), \ldots\)

- **Bad states:**
  - \(SAT(x=0 \land y \neq 0)\)
  - \(SAT(x=1 \land y \neq 1)\)
Invariants

• Arbitrary boolean combination of
  - Equalities and
  - Inequalities
  - Over program quantities

• Note “program quantities” includes variables and induced quantities (like $x^2$)
First Part

- Run tests to get good states

- Run previous algorithm to infer equalities $E$

- Sample bad states
  - Consider `while B do S; assert Q`
  - Sample from $:B \models :Q \models E$
  - Sample from $:B \models WP(assume(B);S,:Q) \models E$
Idea

- **Good and bad states are points in d-dimensional space**
- **Inequalities are planes in this space**
- **Must pick a set of planes that separate every good from every bad state**
• How many planes are required?
  - At most \( m^d \)
  - \( m \) is # points
  - \( d \) is dimensionality

• Puts every point in its own cell
Theorem

• $m^d$ planes (inequalities) would be awful

• **PAC** learning can find a subset of the planes that separate the positive and negative points
  - With $O(s \log m)$ planes
  - Where $s$ is the size of the minimal separator
  - And $m$ is roughly $ds \log ds \ldots$ (other factors) …
  - In time $m^{d+2}$
Simple Example
Disjunction Example
Algorithm

• Consider a bipartite graph
  - Connects every good and bad state

• Repeat
  - Pick a plane cutting the maximum number of remaining edges
Analysis Ingredients

• $m^d$ possible planes

• $s = m^2$ are a separator

• The greedy strategy in time $m^{d+2}$ finds $s \log m$ planes
Comments

• The fact that there is only a log factor increase in number of planes over the minimum is important
  - Avoids overfitting

• In practice, the number of planes is small
Efficiency

• The general algorithm is too inefficient

• Impose some assumptions common to verification techniques
  – Reduce set of candidate planes to polynomial
Predicate Abstraction

• The invariant is an (arbitrary boolean combination) of predicates in $T$

• Can find a PAC separator in time $O(m^2|T|)$
  - Even though the complexity of finding an invariant is $NP^{NP}$ complete
Abstract Interpretation

• Efficient algorithms for restricted abstract domains
  - Boxes $O(m^3d)$
  - Octagons $O(m^3d^2)$
Boxes
Boxes
Check Phase

• Use Boogie

• For counter-examples
  - Satisfies precondition, add as positive example
  - Violates assertion, add as negative example
  - If can’t label, add as a constraint
    • Increases the guess size
## Experiments

<table>
<thead>
<tr>
<th>hsort</th>
<th>47</th>
<th>2</th>
<th>5</th>
<th>0.19</th>
<th>1.05</th>
<th>OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>msort</td>
<td>73</td>
<td>6</td>
<td>10</td>
<td>0.093</td>
<td>1.12</td>
<td>OK</td>
</tr>
<tr>
<td>nested</td>
<td>21</td>
<td>3</td>
<td>4</td>
<td>0.24</td>
<td>0.99</td>
<td>OK</td>
</tr>
<tr>
<td>seq-len1</td>
<td>44</td>
<td>6</td>
<td>5</td>
<td>4.39</td>
<td>1.04</td>
<td>PRE</td>
</tr>
<tr>
<td>seq-len</td>
<td>44</td>
<td>6</td>
<td>5</td>
<td>0.32</td>
<td>1.04</td>
<td>OK</td>
</tr>
<tr>
<td>svd</td>
<td>50</td>
<td>5</td>
<td>5</td>
<td>4.92</td>
<td>0.99</td>
<td>OK</td>
</tr>
<tr>
<td>esc-abs</td>
<td>71</td>
<td>2</td>
<td>6</td>
<td>1.09</td>
<td>1.06</td>
<td>OK</td>
</tr>
<tr>
<td>get-tag</td>
<td>120</td>
<td>2</td>
<td>2</td>
<td>0.092</td>
<td>1.04</td>
<td>OK</td>
</tr>
<tr>
<td>maill-qp</td>
<td>92</td>
<td>1</td>
<td>3</td>
<td>0.11</td>
<td>1.05</td>
<td>OK</td>
</tr>
<tr>
<td>spam</td>
<td>57</td>
<td>2</td>
<td>5</td>
<td>1.01</td>
<td>1.05</td>
<td>OK</td>
</tr>
<tr>
<td>split</td>
<td>20</td>
<td>1</td>
<td>5</td>
<td>FAIL</td>
<td>NA</td>
<td>FAIL</td>
</tr>
<tr>
<td>div</td>
<td>28</td>
<td>1</td>
<td>6</td>
<td>2.03</td>
<td>TO</td>
<td>OK</td>
</tr>
</tbody>
</table>
Application: Equality Checking

- Have extended these techniques to checking equality of arbitrary loops
  - Guess and verify a simulation relation
  - Mine equalities between the two loops as a guide
- Able to prove code generated by gcc -O2 equivalent to CompCert
Discussion

• Sound invariant inference based on PAC learning

• Machine learning/data mining techniques to
  - Handle disjunctions
  - Non-linearities

• Connects complexity of learning and complexity of verification
Discussion

• Like predecessors, focus on numerical invariants
  - Many other interesting aspects of programs not covered
  - Data structures, arrays, concurrency, higher-order functions ...

• This is where we are headed ...
Thanks!

Questions?