Machine-Learning-Guided Adaptive Program Analysis

Hakjoo Oh
Korea University

Kihong Heo
Kwangkeun Yi
Seoul National University

Hongseok Yang
Oxford University

7 September 2016
TAPAS 2016 @ Edinburgh, Scotland
Challenge in Static Analysis

precision

scalability
Challenge in Static Analysis

scalability

precision

?
Challenge in Static Analysis

- precision
- scalability

key: “selectivity”
Flow-Sensitivity

\[ x = y = 0; z = 1 \]

\[ x = z \]

\[ z = z + 1 \]

\[ y = x \]

\[ \text{assert}(y > 0) \]

<table>
<thead>
<tr>
<th></th>
<th>[0,0]</th>
<th>[1,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,0]</td>
<td>[0,0]</td>
<td>[1,1]</td>
</tr>
<tr>
<td>[0,0]</td>
<td>[1,1]</td>
<td>[1,1]</td>
</tr>
<tr>
<td>[1,1]</td>
<td>[2,2]</td>
<td>[2,2]</td>
</tr>
</tbody>
</table>

precise but costly
Flow-Insensitivity

\[
x = y = 0; z = 1
\]

\[
x = z
\]

\[
z = z + 1
\]

\[
y = x
\]

\[
\text{assert}(y > 0)
\]

\[
\begin{array}{|c|c|}
  \hline
  x & [0, +\infty] \\
  \hline
  y & [0, +\infty] \\
  \hline
  z & [1, +\infty] \\
  \hline
\end{array}
\]

cheap but imprecise
Selective Flow-Sensitivity

\[ x = y = 0; z = 1 \]

\[ x = z \]

\[ z = z + 1 \]

\[ y = x \]

assert(y > 0)

**FS : \{x, y\}**

| \( x \) | [0,0] |
| \( y \) | [0,0] |

| \( x \) | [1,\( +\infty \)] |
| \( y \) | [0,0] |

| \( x \) | [1,\( +\infty \)] |
| \( y \) | [0,0] |

**FI : \{z\}**

| \( z \) | [1,\( +\infty \)] |
Selective Flow-Sensitivity

\[
\begin{align*}
x &= y = 0; \quad z = 1 \\
x &= z \\
z &= z + 1 \\
y &= x \\
\text{assert}(y > 0)
\end{align*}
\]

FS : \{y, z\}

<table>
<thead>
<tr>
<th>y</th>
<th>[0,0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>[1,1]</td>
</tr>
</tbody>
</table>

FI : \{x\}

| x  | [0, +\infty] |

\[
\begin{align*}
\text{assert}(y > 0) & \quad \text{fail to prove}
\end{align*}
\]
Hard Search Problem

• Intractably large space, if not infinite
  • $2^{\text{Var}}$ different abstractions for FS
• Most of them are too imprecise or costly
  • $P\left(\{x,y,z\}\right) = \emptyset, \{x\}, \{y\}, \{z\}, \{x,y\}, \{y,z\}, \{x,z\}, \{x,y,z\}$
Our Research

• How to automatically find a good abstraction?
  • pre-analysis [PLDI’14, TOPLAS’16]
  • machine learning techniques [OOPSLA’15, SAS’16, APLAS’16]
Our Learning Approaches

• Learning via black-box optimization [OOPSLA’15]
• Learning via white-box optimization [APLAS’16]
• Learning from automatically labelled data [SAS’16]
• Learning with automatically generated features (in progress)
• ...

Static Analyzer

\[ F(p, a) \Rightarrow n \]

number of proved assertions

abstraction (e.g., a set of variables)
Our Learning Approach
Our Learning Approach

- Parameterized adaptation strategy

\[ S_w : \text{pgm} \rightarrow 2^{\text{Var}} \]
Our Learning Approach

• Parameterized adaptation strategy

\[ S_w : \text{pgm} \rightarrow 2^\text{Var} \]

• Learn a good parameter \( W \) from existing codebase

\[
\begin{array}{c}
P_1, P_2, \ldots, P_m \\
\Rightarrow \\
\text{Codebase} \\
\Rightarrow W
\end{array}
\]
Our Learning Approach

• Parameterized adaptation strategy

\[ S_w : \text{pgm} \rightarrow 2^\text{Var} \]

• Learn a good parameter \( W \) from existing codebase

\[
\begin{align*}
P_1, P_2, \ldots, P_m \\
\phantom{P_1, P_2, \ldots, P_m} \rightarrow W
\end{align*}
\]

• For new program \( P \), run static analysis with \( S_w(P) \)
1. Parameterized Strategy

$S_w : \text{pgm} \rightarrow 2^{\text{Var}}$

(1) Represent program variables as feature vectors.

(2) Compute the score of each variable.

(3) Choose the top-k variables based on the score.
(1) Features

- Predicates over variables:

  \[ f = \{f_1, f_2, \ldots, f_5\} \quad (f_i : \text{Var} \rightarrow \{0, 1\}) \]

- 45 simple syntactic features for variables: e.g,
  - local / global variable, passed to / returned from malloc, incremented by constants, etc
(1) Features

• Represent each variable as a feature vector:

\[ f(x) = \langle f_1(x), f_2(x), f_3(x), f_4(x), f_5(x) \rangle \]

\[ f(x) = \langle 1,0,1,0,0 \rangle \]
\[ f(y) = \langle 1,0,1,0,1 \rangle \]
\[ f(z) = \langle 0,0,1,1,0 \rangle \]
(2) Scoring

- The parameter $w$ is a real-valued vector: e.g.,

\[ w = \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle \]

- Compute scores of variables:

\[
\begin{align*}
\text{score}(x) &= \langle 1,0,1,0,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.3 \\
\text{score}(y) &= \langle 1,0,1,0,1 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.6 \\
\text{score}(z) &= \langle 0,0,1,1,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.1
\end{align*}
\]
Choose the top-k variables based on their scores: e.g., when $k=2$,

- $\text{score}(x) = 0.3$
- $\text{score}(y) = 0.6$
- $\text{score}(z) = 0.1$

In experiments, we chosen 10% of variables with highest scores.
2. Learn a Good Parameter

- Solve the optimization problem:

\[
P_1, P_2, \ldots, P_m \implies W
\]

\[
\text{Codebase}
\]

- Solve the optimization problem:

Find \( w \) that maximizes

\[
\sum_{P_i} F(P_i, S_w(P_i))
\]
Learning via Random Sampling

repeat N times

pick \( \mathbf{w} \in \mathbb{R}^n \) randomly

evaluate \( \sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i)) \)

return best \( \mathbf{w} \) found
Learning via Random Sampling

Table 4. Effectiveness of our method for flow-sensitivity. prove: the number of proved queries in each analysis (FI: flow-insensitivity, FS: flow-sensitivity, partial FS: partial flow-sensitivity). quality: the ratio of proved queries among the queries that require flow-sensitivity. cost: cost increase compared to the FI analysis.

<table>
<thead>
<tr>
<th>Trial</th>
<th>prove</th>
<th>quality</th>
<th>sec</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6,383</td>
<td>80.3 %</td>
<td>2,788</td>
<td>1.6 x</td>
</tr>
<tr>
<td>2</td>
<td>5,788</td>
<td>72.4 %</td>
<td>3,383</td>
<td>79.4 x</td>
</tr>
<tr>
<td>3</td>
<td>6,148</td>
<td>76.3 %</td>
<td>3,023</td>
<td>108.8 x</td>
</tr>
<tr>
<td>4</td>
<td>6,138</td>
<td>73.7 %</td>
<td>3,033</td>
<td>42.0 x</td>
</tr>
<tr>
<td>5</td>
<td>7,343</td>
<td>84.4 %</td>
<td>1,828</td>
<td>20.5 x</td>
</tr>
</tbody>
</table>

| TOTAL  | 31,800| 84.0 %  | 14,055| 17.8 x |

Table 5. Effectiveness for Flow-sensitivity + Context-sensitivity.

By modifying the baseline analyser, we implemented a partially flow-sensitive analyser, which controls its flow-sensitivity according to a given set of abstract locations (program variables, structure fields and allocation sites) as described in Section 6.1. We also implemented our learning algorithm based on Bayesian optimisation.

Our implementations were tested against 30 open source programs from GNU and Linux packages (Table 6 in Appendix).

The key questions that we would like to answer in our experiments are whether our learning algorithm produces a good adaptation strategy and how much it gets benefited from Bayesian optimisation. To answer the first question, we followed a standard method in the machine learning literature, called cross validation. We randomly divide the 30 programs, which supports the full C language and has been being developed for the past seven years [19]. This baseline analyser tracks both numeric and pointer-related information simultaneously in its fixpoint computation. For numeric values, it uses the interval abstract domain, and for pointer values, it uses an allocation-site-based heap abstraction. The analysis is field-sensitive (i.e., separates different structure fields) and flow-sensitive, but it is not context-sensitive. We applied the sparse analysis technique [20] to improve the scalability.

By modifying the baseline analyser, we implemented a partially flow-sensitive analyser, which controls its flow-sensitivity according to a given set of abstract locations (program variables, structure fields and allocation sites) as described in Section 6.1. We also implemented our learning algorithm based on Bayesian optimisation.

Our implementations were tested against 30 open source programs from GNU and Linux packages (Table 6 in Appendix).

The key questions that we would like to answer in our experiments are whether our learning algorithm produces a good adaptation strategy and how much it gets benefited from Bayesian optimisation. To answer the first question, we followed a standard method in the machine learning literature, called cross validation. We randomly divide the 30 programs, which supports the full C language and has been being developed for the past seven years [19]. This baseline analyser tracks both numeric and pointer-related information simultaneously in its fixpoint computation. For numeric values, it uses the interval abstract domain, and for pointer values, it uses an allocation-site-based heap abstraction. The analysis is field-sensitive (i.e., separates different structure fields) and flow-sensitive, but it is not context-sensitive. We applied the sparse analysis technique [20] to improve the scalability.
Bayesian Optimization

• A powerful method for solving difficult black-box optimization problems.

• Especially powerful when the objective function is expensive to evaluate.

• Key idea: use a probabilistic model to reduce the number of objective function evaluations.
Learning via Bayesian Optimization

repeat N times

select a promising \( w \) using the model

evaluate \( \sum_{P_i} F(P_i, S_w(P_i)) \)

update the probabilistic model

return best \( w \) found

- Probabilistic model: Gaussian processes
- Selection strategy: Expected improvement
Learning via Bayesian Optimization

![Graph showing distribution of quality and count. The x-axis represents quality ranging from 0 to 80, and the y-axis represents count ranging from 0 to 300. There is a peak in the distribution near quality 70, indicating a higher count of some quality metric at that point.]

Our implementations were tested against 30 open source programs from GNU and Linux packages (Table 6 in Appendix). The key questions that we would like to answer in our experiments are whether our learning algorithm produces a good adaptation strategy and how much it gets benefited from Bayesian optimisation. To answer the first question, we followed a standard method in the machine learning literature, called cross validation. We randomly divide the 30 programs into training and testing sets.

The implementation of our learning algorithm is available at [link] (http://prl.korea.ac.kr/~hakjoo/research/oopsla15/).
Effectiveness

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source programs
  - 20 for training, 10 for testing
Effectiveness

• Implemented in Sparrow, an interval analyzer for C
• Evaluated on 30 open-source programs
  • 20 for training, 10 for testing

Precision

<table>
<thead>
<tr>
<th>FI</th>
<th>SFS</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>70</td>
<td>100</td>
</tr>
</tbody>
</table>
Effectiveness

• Implemented in Sparrow, an interval analyzer for C
• Evaluated on 30 open-source programs
  • 20 for training, 10 for testing

Precision

<table>
<thead>
<tr>
<th>Fl</th>
<th>SFS</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>70</td>
<td>100</td>
</tr>
</tbody>
</table>

Cost

<table>
<thead>
<tr>
<th>Fl</th>
<th>SFS</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>2x</td>
<td>18x</td>
</tr>
</tbody>
</table>
Limitations

• While promising, the method has limitations:
  • black-box optimization is inherently inefficient
  • manual feature engineering is needed

• Follow-up work to overcome the limitations:
  • improving the efficiency [APLAS’16, SAS’16]
  • automating feature engineering [on-going]
Improving Efficiency

• A white-box optimization method [APLAS’16]

\[
\mathcal{O}_P : \mathbb{J}_P \rightarrow \mathbb{R}.
\]

Find \( w^* \) that minimizes \( \sum_{j \in \mathbb{J}_P} (score^w_P(j) - \mathcal{O}(j))^2 \)

• A supervised learning method [SAS’16]
## Manual Feature Engineering

- The success of ML heavily depends on the “features”
- Feature engineering is nontrivial and time-consuming
- Features do not generalize to other analyses

<table>
<thead>
<tr>
<th>Type</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>local variable</td>
</tr>
<tr>
<td>B</td>
<td>global variable</td>
</tr>
<tr>
<td>C</td>
<td>structure field</td>
</tr>
<tr>
<td>D</td>
<td>location created by dynamic memory allocation</td>
</tr>
<tr>
<td>E</td>
<td>assigned a constant expression (e.g., ( x = 1 + z ))</td>
</tr>
<tr>
<td>F</td>
<td>compared with a constant expression (e.g., ( x &lt; 0 ))</td>
</tr>
<tr>
<td>G</td>
<td>compared with an other variable (e.g., ( x &lt; y ))</td>
</tr>
<tr>
<td>H</td>
<td>located in a conditional expression (e.g., if( x ))</td>
</tr>
<tr>
<td>I</td>
<td>directly used in malloc (e.g., malloc( x ))</td>
</tr>
<tr>
<td>J</td>
<td>indirectly used in malloc (e.g., y = x; malloc( x ))</td>
</tr>
<tr>
<td>K</td>
<td>directly used in realloc (e.g., realloc( x ))</td>
</tr>
<tr>
<td>L</td>
<td>indirectly used in realloc (e.g., y = x; realloc( y ))</td>
</tr>
<tr>
<td>M</td>
<td>directly return an allocated memory</td>
</tr>
<tr>
<td>N</td>
<td>indirectly return a constant expression</td>
</tr>
<tr>
<td>O</td>
<td>return expression involves field access</td>
</tr>
<tr>
<td>P</td>
<td>return value depends on a structure field</td>
</tr>
<tr>
<td>Q</td>
<td>return void</td>
</tr>
<tr>
<td>R</td>
<td>directly invoked with a constant</td>
</tr>
<tr>
<td>S</td>
<td>constant zero</td>
</tr>
<tr>
<td>T</td>
<td>pass to a function</td>
</tr>
<tr>
<td>U</td>
<td>pass to a statement</td>
</tr>
<tr>
<td>V</td>
<td>pass to a control flow statement</td>
</tr>
<tr>
<td>W</td>
<td>pass to a basic block</td>
</tr>
<tr>
<td>X</td>
<td>pass to a loop</td>
</tr>
<tr>
<td>Y</td>
<td>pass to a conditional branch</td>
</tr>
<tr>
<td>Z</td>
<td>pass to a return statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>leaf function</td>
</tr>
<tr>
<td>B</td>
<td>function containing malloc</td>
</tr>
<tr>
<td>C</td>
<td>function containing realloc</td>
</tr>
<tr>
<td>D</td>
<td>function containing a loop</td>
</tr>
<tr>
<td>E</td>
<td>function containing an if statement</td>
</tr>
<tr>
<td>F</td>
<td>function containing a switch statement</td>
</tr>
<tr>
<td>G</td>
<td>function using a string-related library function</td>
</tr>
<tr>
<td>H</td>
<td>write to a global variable</td>
</tr>
<tr>
<td>I</td>
<td>write to a structure field</td>
</tr>
<tr>
<td>J</td>
<td>directly return an allocated memory</td>
</tr>
<tr>
<td>K</td>
<td>indirectly return a constant expression</td>
</tr>
<tr>
<td>L</td>
<td>return expression involves field access</td>
</tr>
<tr>
<td>M</td>
<td>return value depends on a structure field</td>
</tr>
<tr>
<td>N</td>
<td>return void</td>
</tr>
<tr>
<td>O</td>
<td>directly invoked with a constant</td>
</tr>
<tr>
<td>P</td>
<td>constant zero</td>
</tr>
<tr>
<td>Q</td>
<td>pass to a function</td>
</tr>
<tr>
<td>R</td>
<td>pass to a statement</td>
</tr>
<tr>
<td>S</td>
<td>pass to a control flow statement</td>
</tr>
<tr>
<td>T</td>
<td>pass to a basic block</td>
</tr>
<tr>
<td>U</td>
<td>pass to a loop</td>
</tr>
<tr>
<td>V</td>
<td>pass to a conditional branch</td>
</tr>
<tr>
<td>W</td>
<td>pass to a return statement</td>
</tr>
</tbody>
</table>

### flow-sensitivity

- used in an array declaration (e.g., \( \ast c \))
- used in a memory allocation (e.g., malloc\( x \))
- used in the right-hand-side of an assignment (e.g., \( x = c \))
- used with the less-than operator (e.g., \( x < c \))
- used with the greater-than operator (e.g., \( x > c \))
- used with the equality operator (e.g., \( x = c \))
- used with the not-equality operator (e.g., \( x \neq c \))
- used within conditional expressions (e.g., \( x < c \))
- used inside loops
- used in return statements (e.g., return \( c \))
- constant zero

### context-sensitivity

- functions having a structure as an argument
- functions having an integer argument
- functions having a string-related library function
- functions containing a switch statement
- functions containing an if statement
- functions using a string-related library function
- functions containing a loop
- functions containing a constant
- functions containing a structure
- functions containing a string
- functions containing a function
- functions containing a control flow statement
- functions containing a basic block
- functions containing a loop
- functions containing a conditional branch
- functions containing a return statement

### widening thresholds

- loss of generality, let us assume that
- the widening.
- that the widening operator extrapolates any unstable bounds simply to infinity. For
- instance, a simple widening operator for the interval domain works as follows: (For
- the abstract interpretation framework guarantees that the above chain is always fi-
- cially defined as follows:

\[
\begin{align*}
&X_{i+1} = \min(\lim\{X_i\}) + min(\{l\}) - lim(\{u\}), \\
&X_{i+2} = \max(\{l\}) + max(\{u\}), \\
&\vdots \\
&X_{i+n} = \max(\{l\}) + max(\{u\})
\end{align*}
\]

For instance,

\[\lim\{X_i\} + \max(\{l\}) - \min(\{u\}) = \max(\{l\}) + \max(\{u\}).\]

Thus, the widening operator parameterized by

\[
\{\min(\phi_i), \max(\phi_i)\}
\]

with

\[
\phi_i = \begin{cases} \min & \text{if } i = 0 \\ \max & \text{otherwise} \end{cases}
\]

plus widening thresholds for the interprocedural widening.

### Table II

<table>
<thead>
<tr>
<th>Type</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>local variable</td>
</tr>
<tr>
<td>B</td>
<td>global variable</td>
</tr>
<tr>
<td>C</td>
<td>structure field</td>
</tr>
<tr>
<td>D</td>
<td>location created by dynamic memory allocation</td>
</tr>
<tr>
<td>E</td>
<td>assigned a constant expression (e.g., ( x = 1 + z ))</td>
</tr>
<tr>
<td>F</td>
<td>compared with a constant expression (e.g., ( x &lt; 0 ))</td>
</tr>
<tr>
<td>G</td>
<td>compared with an other variable (e.g., ( x &lt; y ))</td>
</tr>
<tr>
<td>H</td>
<td>located in a conditional expression (e.g., if( x ))</td>
</tr>
<tr>
<td>I</td>
<td>directly used in malloc (e.g., malloc( x ))</td>
</tr>
<tr>
<td>J</td>
<td>indirectly used in malloc (e.g., y = x; malloc( x ))</td>
</tr>
<tr>
<td>K</td>
<td>directly used in realloc (e.g., realloc( x ))</td>
</tr>
<tr>
<td>L</td>
<td>indirectly used in realloc (e.g., y = x; realloc( y ))</td>
</tr>
<tr>
<td>M</td>
<td>directly return an allocated memory</td>
</tr>
<tr>
<td>N</td>
<td>indirectly return a constant expression</td>
</tr>
<tr>
<td>O</td>
<td>return expression involves field access</td>
</tr>
<tr>
<td>P</td>
<td>return value depends on a structure field</td>
</tr>
<tr>
<td>Q</td>
<td>return void</td>
</tr>
<tr>
<td>R</td>
<td>directly invoked with a constant</td>
</tr>
<tr>
<td>S</td>
<td>constant zero</td>
</tr>
<tr>
<td>T</td>
<td>pass to a function</td>
</tr>
<tr>
<td>U</td>
<td>pass to a statement</td>
</tr>
<tr>
<td>V</td>
<td>pass to a control flow statement</td>
</tr>
<tr>
<td>W</td>
<td>pass to a basic block</td>
</tr>
<tr>
<td>X</td>
<td>pass to a loop</td>
</tr>
<tr>
<td>Y</td>
<td>pass to a conditional branch</td>
</tr>
<tr>
<td>Z</td>
<td>pass to a return statement</td>
</tr>
</tbody>
</table>

### Table IV

<table>
<thead>
<tr>
<th>Type</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>local variable</td>
</tr>
<tr>
<td>B</td>
<td>global variable</td>
</tr>
<tr>
<td>C</td>
<td>structure field</td>
</tr>
<tr>
<td>D</td>
<td>location created by dynamic memory allocation</td>
</tr>
<tr>
<td>E</td>
<td>assigned a constant expression (e.g., ( x = 1 + z ))</td>
</tr>
<tr>
<td>F</td>
<td>compared with a constant expression (e.g., ( x &lt; 0 ))</td>
</tr>
<tr>
<td>G</td>
<td>compared with an other variable (e.g., ( x &lt; y ))</td>
</tr>
<tr>
<td>H</td>
<td>located in a conditional expression (e.g., if( x ))</td>
</tr>
<tr>
<td>I</td>
<td>directly used in malloc (e.g., malloc( x ))</td>
</tr>
<tr>
<td>J</td>
<td>indirectly used in malloc (e.g., y = x; malloc( x ))</td>
</tr>
<tr>
<td>K</td>
<td>directly used in realloc (e.g., realloc( x ))</td>
</tr>
<tr>
<td>L</td>
<td>indirectly used in realloc (e.g., y = x; realloc( y ))</td>
</tr>
<tr>
<td>M</td>
<td>directly return an allocated memory</td>
</tr>
<tr>
<td>N</td>
<td>indirectly return a constant expression</td>
</tr>
<tr>
<td>O</td>
<td>return expression involves field access</td>
</tr>
<tr>
<td>P</td>
<td>return value depends on a structure field</td>
</tr>
<tr>
<td>Q</td>
<td>return void</td>
</tr>
<tr>
<td>R</td>
<td>directly invoked with a constant</td>
</tr>
<tr>
<td>S</td>
<td>constant zero</td>
</tr>
<tr>
<td>T</td>
<td>pass to a function</td>
</tr>
<tr>
<td>U</td>
<td>pass to a statement</td>
</tr>
<tr>
<td>V</td>
<td>pass to a control flow statement</td>
</tr>
<tr>
<td>W</td>
<td>pass to a basic block</td>
</tr>
<tr>
<td>X</td>
<td>pass to a loop</td>
</tr>
<tr>
<td>Y</td>
<td>pass to a conditional branch</td>
</tr>
<tr>
<td>Z</td>
<td>pass to a return statement</td>
</tr>
</tbody>
</table>

### Table IV: Features for widening-with-thresholds.

- loss of generality, let us assume that
- the widening.
- that the widening operator extrapolates any unstable bounds simply to infinity. For
- instance, a simple widening operator for the interval domain works as follows: (For
- the abstract interpretation framework guarantees that the above chain is always fi-
- cially defined as follows:

\[
\begin{align*}
&X_{i+1} = \min(\lim\{X_i\}) + min(\{l\}) - lim(\{u\}), \\
&X_{i+2} = \max(\{l\}) + max(\{u\}), \\
&\vdots \\
&X_{i+n} = \max(\{l\}) + max(\{u\})
\end{align*}
\]

For instance,

\[\lim\{X_i\} + \max(\{l\}) - \min(\{u\}) = \max(\{l\}) + \max(\{u\}).\]

Thus, the widening operator parameterized by

\[
\{\min(\phi_i), \max(\phi_i)\}
\]

with

\[
\phi_i = \begin{cases} \min & \text{if } i = 0 \\ \max & \text{otherwise} \end{cases}
\]

practical experience.

### References

- Lee et al. (2023, Accepted for ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

---

1. manual feature engineering is nontrivial and time-consuming.
2. features do not generalize to other analyses.
Automating Feature Engineering

Before [OOPSLA’15, SAS’16, APLAS’16]

New method
Key Ideas

• Use a program reducer to generate feature programs that capture the key reason why FS succeeds but FI fails.

```c
int j = 0;
main() {
    j++;
    assert (j>0);
}
```

• Generalize the programs by abstract data flow graphs

![Diagram showing abstract data flow graphs]

![Diagram showing reduced program and feature comparison]
Summary

• Challenges in selective static analysis
• Using machine learning is promising
  • [OOPSLA’15, SAS’16, APLAS’16,…]
  • flow-sensitivity, context-sensitivity, relational domain, widening thresholds, soundness, etc
• Generally applicable beyond static analysis
  • e.g., concolic testing
Summary

• Challenges in selective static analysis
• Using machine learning is promising
  • [OOPSLA’15, SAS’16, APLAS’16,…]
  • flow-sensitivity, context-sensitivity, relational domain, widening thresholds, soundness, etc
• Generally applicable beyond static analysis
  • e.g., concolic testing

Thank you