Machine Learning Approaches to Selective Program Analyses

오학주
고려대학교 프로그래밍 연구실
(with 채권수, 홍성준, 이민아, 양홍석, 이광근)

Feb. 17, 2016 @SIGPL Winter workshop
Scalability and Precision via Selectivity

Scalability

Precision
Flow-Sensitivity

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

\[ x = z \]

\[ z = z + 1 \]

\[ y = x \]

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

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

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

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

precise but costly
Flow-Insensitivity

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

\[
x = z
\]

\[
z = z + 1
\]

\[
y = x
\]

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

cheap but imprecise

<table>
<thead>
<tr>
<th></th>
<th>([0, +\infty])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>(y)</td>
<td>([0, +\infty])</td>
</tr>
<tr>
<td>(z)</td>
<td>([1, +\infty])</td>
</tr>
</tbody>
</table>
Selective Flow-Sensitivity

\[ x = y = 0; z = 1 \]
\[ x = z \]
\[ z = z + 1 \]
\[ y = x \]
\[ \text{assert}(y > 0) \]

\[ \text{FS} : \{ x, y \} \]

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

\[ \text{FI} : \{ z \} \]

\[ z \quad [1, +\infty] \]
Selective Flow-Sensitivity

FS : \{y,z\}

FI : \{x\}

fail to prove
Hard Search Problem

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

• How to efficiently find a good abstraction?

• Two directions:
  • PL approaches [PLDI’14, TOPLAS’16]
  • ML approaches [OOPSLA’15, on-going work]
Learning-based Approach

• Parameterized adaptation strategy

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

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

\[ \begin{array}{c}
\text{P}_1, \text{P}_2, \ldots, \text{P}_m \\
\text{Codebase}
\end{array} \Rightarrow W \]

• 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

• Represent each variable as a feature vector:

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

• The parameter $\mathbf{w}$ is a real-valued vector: e.g.,

$$\mathbf{w} = \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle$$

• Compute scores of variables:

$$\text{score}(x) = \langle 1,0,1,0,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.3$$
$$\text{score}(y) = \langle 1,0,1,0,1 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.6$$
$$\text{score}(z) = \langle 0,0,1,1,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.1$$
(3) Choose Top-k Variables

• Choose the top-k variables based on their scores: e.g., when k=2,

\[
\begin{align*}
\text{score}(x) &= 0.3 \\
\text{score}(y) &= 0.6 \\
\text{score}(z) &= 0.1
\end{align*}
\]

\[
\{x, y\}
\]

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

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

- Solve the optimization problem:

\[ \text{Find } w \text{ that maximizes } \sum_{P_i} F(P_i, S_w(P_i)) \]
Solving the Opt. Problem

- How to solve the optimization problem efficiently?

\[ \text{Find } w \text{ that maximizes } \sum_{P_i} F(P_i, S_w(P_i)) \]

- Using ideas of Bayesian optimization and ordinal optimization
Effectiveness

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source benchmarks

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>

Cost

<table>
<thead>
<tr>
<th>FI</th>
<th>SFS</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>2x</td>
<td>18x</td>
</tr>
</tbody>
</table>
• The success crucially depends on the choice of features
• Feature construction is nontrivial and tedious
• |analyzers| x |parameter types| x |query types|
Learning with Automatic Feature Construction

- Fully automatic learning approach

\[ P_1, P_2, \ldots, P_m \rightarrow \text{Features} \]

\[ \text{Training Examples} \rightarrow \mathcal{C} : P \rightarrow B \]
Key Ideas

• Generate *feature programs* that capture the key reason why FS succeeds but FI fails.

```c
int j = 0;
main() {
    char num[5];
    int tmp = j++;
    num[tmp];
}
```

• Apply FS to a new program, if it matches some feature program.

```c
double B[309];
main() {
    for (int i=1; i<50; i++) {
        B[i];
    }
}
```
Key Ideas

• Feature programs are automatically generated by using a general-purpose program reducer.

\[
\text{reduce} : \mathcal{P} \times (\mathcal{P} \rightarrow \mathbb{B}) \rightarrow \mathcal{P}
\]

• Keep reducing when FS succeeds but FI fails:

\[
\phi(P) = F(P, 0) = 0 \land F(P, 1) = 1
\]
cf) Other Applications

• Bug-finding of static analyzers

• Alarm reduction:

\[
\begin{align*}
  a &= \text{input}(); \\
  b &= a; \\
  10 \ / \ b;
\end{align*}
\]

\[
\begin{align*}
  a = \alpha \wedge b = a \wedge b \neq 0 \\
  \iff \\
  a = \alpha \wedge a \neq 0
\end{align*}
\]
Summary

• Key problem in static analysis: automatic adaptation
• Promising approach: use ML [OOPSLA’15]
• Major hurdle: manual feature construction
• Our Solution: generate and match feature programs

Thank you