머신러닝 기반 선별적 프로그램 분석

Machine Learning-Guided Adaptive Program Analysis

오학주
고려대학교 정보대학 컴퓨터학과
(with 허기홍, 채권수, 차수영, 정세훈, 홍성준, 양홍석, 이광근)

Aug. 19, 2016 @SIGPL Summer School
Static Program Analysis

- Predict program behavior statically and automatically
  - static: before execution, at compile-time
  - automatic: sw is analyzed by sw (“static analyzers”)
- Applications
  - bug-finding: e.g., find runtime failures of programs
  - security: e.g., is this app malicious or benign?
  - verification: e.g., does the program meet its specification?
  - compiler optimization, e.g., automatic parallelization
Principle of Program Analysis

sound vs. unsound

program states

error states

program states

error states
Principle of Program Analysis

- Imprecise
  - Program states
  - Error states
  - False alarms

- Precise
  - Program states
  - Error states
Principle of Program Analysis
Challenge in Static Analysis

scalability

precision

key: “selectivity”
Flow-Sensitivity

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

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

precise but costly

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

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
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>x $[0, +\infty]$</th>
<th>y $[0, +\infty]$</th>
<th>z $[1, +\infty]$</th>
</tr>
</thead>
</table>
Selective Flow-Sensitivity

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

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

<table>
<thead>
<tr>
<th></th>
<th>[0,0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>[0,0]</td>
</tr>
<tr>
<td>(y)</td>
<td>[0,0]</td>
</tr>
</tbody>
</table>

\[ x = [1, +\infty] \]
\[ y = [0,0] \]

**FI : \{z\}**

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

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

**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] |

fail to prove
Context-Sensitivity

```c
int h(n) {ret n;}

void f(a) {
    x = h(a);
    assert(x > 1); // Q1 always holds
    y = h(input());
    assert(y > 1); // Q2 does not always hold
}

c3: void g() {f(8);}  

void m() {
    f(4);
    g();
    g();
    g();
}
```
int h(n) { ret n; }

void f(a) {
  x = h(a);
  assert(x > 1);  // Q1
  y = h(input());
  assert(y > 1);  // Q2
}

c3: void g() { f(8); }

c4: f(4);
c5: g();
c6: g();

precise but costly
Context-Insensitivity

```c
int h(n) {ret n;}

void f(a) {
  x = h(a);
  assert(x > 1);  // Q1
  y = h(input());
  assert(y > 1);  // Q2
}

c3: void g() {f(8);}

c5, c6: g();

c3: g();

c1, c2: cheap but imprecise
```
int h(n) { ret n; }

void f(a) {
  x = h(a);
  assert(x > 1); // Q1
  y = h(input());
  assert(y > 1); // Q2
}

c3: void g() { f(8); }

void m() {
  f(4);
  g();
  g();
}
How to select?

• Often done manually by analysis designers

• Finding a good selection strategy is an art:
  
  • Intractably large space, if not infinite: ex) $2^{\text{Var}}$ different abstractions for FS
  
  • Most of them are too imprecise or costly ex) $P(\{x,y,z\}) = \emptyset, \{x\}, \{y\}, \{z\}, \{x,y\}, \{y,z\}, \{x,z\}, \{x,y,z\}$
Our Research

- Develop techniques for automatically finding the selection strategies
  - [PLDI’14, OOPSLA’15, TOPLAS’16, SAS’16, APLAS’16]
- Use machine learning techniques to learn a good strategy from freely available data.
Contents

• 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)
• Learning unsoundness strategy (in progress)
• Learning search strategy of concolic testing (in progress)
• Learning static analyzers (in progress)
Learning via Blackbox Optimization (OOPSLA'15)
Static Analyzer

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

- **abstraction** (e.g., a set of variables)
- **number of proved assertions**
Overall Approach

- Parameterized adaptation strategy
  
  \[ S_w : \text{pgm} \rightarrow 2^{\text{Var}} \]

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

\[ \text{Codebase} \quad \Rightarrow \quad W \]

- 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

- Solve the optimization problem:

$$P_1, P_2, \ldots, P_m \quad \Rightarrow \quad W$$

Codebase

- Find $w$ that maximizes

$$\sum_{P_i} F(P_i, S_w(P_i))$$
repeat N times

pick $w \in \mathbb{R}^n$ randomly

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

return best $w$ found
Learning via Random Sampling

![Graph showing count vs. quality distribution](image)

**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>FI</th>
<th>FS</th>
<th>partial FS</th>
<th>FI</th>
<th>FS</th>
<th>partial FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6,383</td>
<td>9,237</td>
<td>8,674</td>
<td>80.3%</td>
<td>2,788</td>
<td>46</td>
</tr>
<tr>
<td>2</td>
<td>5,788</td>
<td>8,287</td>
<td>7,598</td>
<td>72.4%</td>
<td>3,383</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>6,148</td>
<td>8,737</td>
<td>8,123</td>
<td>76.3%</td>
<td>3,023</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>6,138</td>
<td>9,883</td>
<td>8,899</td>
<td>73.7%</td>
<td>3,033</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>7,343</td>
<td>10,082</td>
<td>10,040</td>
<td>98.5%</td>
<td>1,828</td>
<td>30</td>
</tr>
<tr>
<td>TOTAL</td>
<td>31,800</td>
<td>46,226</td>
<td>43,334</td>
<td>80.0%</td>
<td>14,055</td>
<td>219</td>
</tr>
</tbody>
</table>

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

(a) Random sampling (b) Bayesian optimisation

Figure 2. Comparison of Bayesian optimisation with random sampling
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 with count on the y-axis and quality on the x-axis. The graph has a peak at a quality value around 55, with a long tail to the right.]
Effectiveness

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

**Precision**

- FI
- SFS
- FS

Precision values:
- FI: 0
- SFS: 70
- FS: 100

**Cost**

- FI
- SFS
- FS

Cost values:
- FI: 1x
- SFS: 2x
- FS: 18x
Automatically Generating Features (In Progress)
The success of ML heavily depends on the “features”

Feature engineering is nontrivial and time-consuming

Features do not generalize to other domains
Automatic Feature Generation

Before

Codebase $\rightarrow$ Hand-crafted features $\rightarrow$ Parameter values $\rightarrow$ Adaptation Strategy

New method

Codebase $\rightarrow$ Features $\rightarrow$ Parameter values $\rightarrow$ Adaptation Strategy

(analogous to representation learning, deep learning, etc in ML)
Example: Flow-Sensitive Analysis

- A query-based, partially flow-sensitive interval analysis
- The analysis uses a query-classifier \( C : \text{Query} \to \{1,0\} \)

```plaintext
x = 0; y = 0; z = input(); w = 0;
y = x; y++;
assert (y > 0); // Query 1 provable
assert (z > 0); // Query 2 unprovable
assert (w == 0); // Query 3 unprovable
```

<table>
<thead>
<tr>
<th>line</th>
<th>flow-sensitive result</th>
<th>flow-insensitive result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>abstract state</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( { x \mapsto [0,0], y \mapsto [0,0] } )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( { x \mapsto [0,0], y \mapsto [1,1] } )</td>
<td>( { z \mapsto [0,0], w \mapsto [0,0] } )</td>
</tr>
<tr>
<td>3</td>
<td>( { x \mapsto [0,0], y \mapsto [1,1] } )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( { x \mapsto [0,0], y \mapsto [1,1] } )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( { x \mapsto [0,0], y \mapsto [1,1] } )</td>
<td></td>
</tr>
</tbody>
</table>
Learning a Query Classifier

Standard binary classification:

\[ \{(q_i, b_i)\}_{i=1}^{n} \quad \rightarrow \quad \{(v_i, b_i)\}_{i=1}^{n} \quad \rightarrow \quad C : \mathbb{B}^k \rightarrow \mathbb{B} \]

\( v_i \in \mathbb{B}^k \)

- Feature extraction is a key to success
- Raw data should be converted to suitable representations from which classification algorithms could find useful patterns

We aim to automatically find the right representation
Feature Extraction

- Features and matching algorithm:
  - a set of features: $\Pi = \{\pi_1, \ldots, \pi_k\}$
  - match : $Query \times Feature \rightarrow \mathbb{B}$
- Transform the query $q$ into the feature vector:
  $$\langle \text{match}(q, \pi_1), \ldots, \text{match}(q, \pi_k) \rangle$$
Generating Features

\[ \Pi = \{ \pi_1, \ldots, \pi_k \} \]

- A feature is a graph that describes data flows of queries
- What makes good features?
  - *selective* to key aspects for discrimination
  - *invariant* to irrelevant aspects for generalization
- Generating features:
  - Generate *feature programs* by running reducer
  - Represent the feature programs by data-flow graphs
- \( \Pi \) is the set of all data flow graphs generated from the codebase
Generating Features

• Feature program $P$ is a minimal program such that

$$\phi(P) \equiv FI(P) = \text{unproven} \land FS(P) = \text{proven}$$

• Generic program reducer: e.g., C-Reduce [PLDI’12]

$$\text{reduce} : P \times (P \rightarrow B) \rightarrow P$$

• Reducing programs while preserving the condition

$$\text{reduce}(P, \phi)$$

generates feature programs.
Generating Features

- Reduce programs while preserving the condition

\[ \phi(P) \equiv FI(P) = \text{unproven} \land FS(P) = \text{proven} \]

```plaintext
a = 0; b = 0;
while (1) {
    b = unknown();
    if (a > b) reduce(P, \phi) =>
        if (a < 3)
            assert (a < 5);
    a++;
}
```

```plaintext
a = 0;
while (1) {
    if (a < 3)
        assert (a < 5);
    a++;
}
```
### Generating Features

- Represent the features by abstract data flow graphs

```c
1 a = 0;
2 while (1) {
3   if (a < 3)  
4     assert (a < 5);
5   a++;
6  }
```

- The right level of abstraction is learned from codebase
Matching Algorithm

\[ \text{match : Query} \times \text{Feature} \rightarrow \mathbb{B} \]

Subgraph inclusion:

\[
(N_1, E_1) \subseteq (N_2, E_2) \iff N_1 \subseteq N_2 \land E_1 \subseteq E_2^* \]

Example program and feature

```
1  a = 0; b = 0;
2  while (1) {
3     b = unknown();
4     if (a > b)
5        if (a < 3)
6           assert (a < 5);
7           a++;
8   }
```

```
id := id + c
id := c \rightarrow id < c \rightarrow Q(id < c)
```

```
id := \top
id > id
id := id + c
id := c \rightarrow id < c \rightarrow Q(id < c)
```
### Performance

- Partially flow-sensitive interval analysis
- Partially relational octagon analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Selection</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>1</td>
<td>59.8 %</td>
<td>71.2 %</td>
</tr>
<tr>
<td>2</td>
<td>70.3 %</td>
<td>92.0 %</td>
</tr>
<tr>
<td>3</td>
<td>68.0 %</td>
<td>90.3 %</td>
</tr>
<tr>
<td>4</td>
<td>82.8 %</td>
<td>72.7 %</td>
</tr>
<tr>
<td>5</td>
<td>68.1 %</td>
<td>67.1 %</td>
</tr>
<tr>
<td>TOTAL</td>
<td>70.5 %</td>
<td>81.5 %</td>
</tr>
</tbody>
</table>
Other PA + ML Approaches
Learning via White-box Optimization [APLAS’16]

• The black-box optimization method is too slow when the codebase is large

• Replace it to an easy-to-solve white-box problem by using oracle:
  \[ O_P : J_P \rightarrow \mathbb{R}. \]

Find \( w^* \) that minimizes
  \[
  \sum_{j \in J_P} (score_P^w(j) - O(j))^2
  \]

• Oracle is obtained from a single run of codebase

• 26x faster to learn a comparable strategy
Learning from Automatically Labelled Data [SAS’16]

• Learning a variable clustering strategy for Octagon is too difficult to solve with black-box optimization

• Replace it to a (much easier) supervised-learning problem:

<table>
<thead>
<tr>
<th>a</th>
<th>-a</th>
<th>b</th>
<th>-b</th>
<th>c</th>
<th>-c</th>
<th>i</th>
<th>-i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>-a</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>-b</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>-c</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>-i</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

• Who label the data? by impact pre-analysis [PLDI’14].

• The ML-guided Octagon analysis is 33x faster than the pre-analysis-guided one with 2% decrease in precision.
Learning Unsoundness Strategies (in submission)

- **uniformly sound**: FP: 78%
- **selectively unsound**: FP: 23%, FN: 13%
- **uniformly unsound**: FP: 23%, FN: 85%
Data-Driven Concolic Testing (in progress)

- The efficacy of concolic testing heavily depends on the search strategy

\[ S \in \text{Strategy} = \text{Path} \rightarrow \text{Branch} \]

- Search strategies are manually designed (heuristics)
  - e.g., \( S_{\text{rand}}, S_{\text{dfs}}, S_{\text{cfg}}, S_{\text{cgs}}, \ldots \)
  - a huge amount of engineering efforts
  - sub-optimal performance

- Automate the process: \( S_\theta : \text{Path} \rightarrow \text{Branch} \)

Find \( \theta^* \) that maximizes \( \sum_{P_i \in \mathbb{P}} C(P_i, S_{\theta^*}) \)
Learning Static Analyzers (in progress)

• The usage of static analyzers is limited in extreme (yet daily in practice) situations:
  • It cannot analyze unparsable programs.
  • It does not scale to the entire Linux package.
  • A C analyzer cannot be used even for C++ code.
  • A source code analyzer cannot be used for binary code.
Summary

- Adaptation is a key problem in static analysis
- Using ML is a promising and exciting direction
- Something is done with hand-tuning?
  - Parameterize it
  - Learn the best parameters from data

Thank you