Learning to Synthesize

Yingfei Xiong
Peking University
SAVE 2019
Bug Fixing Costs a Lot

• Developers spend 50% of their time debugging\textsuperscript{[1]}
• The development team often does not have enough resource for bug-fixing\textsuperscript{[2]}
• Software is often released with known bugs\textsuperscript{[3]}

\textsuperscript{[1]} Britton et al. Quantify the time and cost saved using reversible debuggers. Cambridge report, 2013
\textsuperscript{[2]} J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” eXchange, 2005, pp. 35–39
Automated Program Repair

Program Specification

Automated Program Repair System

High-Quality Patches

Developer: Repair Quality↑↑ Repair Efficiency↑ [1]

Weak Specification Problem

• Programs usually have only weak specification such as tests.

• Early systems aim to meet the specification, often producing low-quality patches.

Precisions of some popular systems (before 2016)

How to deal with the weak specification?

• Find the most-likely patch under the current context

• Precisions of Recent tools:
  • ACS\textsuperscript{[1]} + Patch Filtering\textsuperscript{[2]} : 85%
  • ConCap\textsuperscript{[3]} : 84%

• This talk:
  • A generalization of this weak specification problem
  • A general framework to address this problem

\textsuperscript{[1]} Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, Lu Zhang. Precise Condition Synthesis for Program Repair. ICSE'17.
\textsuperscript{[2]} Yingfei Xiong, Xinyuan Liu#, Muhan Zeng#, Lu Zhang, Gang Huang. Identifying Patch Correctness in Test-Based Program Repair. ICSE'18.
\textsuperscript{[3]} Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, Shing-Chi Cheung: Context-aware patch generation for better automated program repair. ICSE'18.
Program Estimation

• We aim to find the program that are most-likely to be written under the current context.

```java
public static long factorial(final int n){
    if( ... ){  } } } 
```

• We define this problem as **program estimation**:  
  • Given a context `context`, a (weak) specification `spec`, and a space of programs `Prog`, find program  
    
    \[
    prog = \arg\max_{prog \in Prog \land prog \vdash spec} P(prog \mid context)
    \]

• A sub-problem of program synthesis
Application: Test-based Program Repair

• Context = buggy program & at least one failed test

Buggy code

```java
/** Compute the maximum of two values
 * @param a first value
 * @param b second value
 * @return b if a is lesser or equal to b, a otherwise
 */
public static int max(final int a, final int b) {
    return (a <= b) ? a : b;
}
```
Application: Code Completion

• Context = partial code

```java
public static long fibonacci(int n) {
    if ( ?? ) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```
Application: Program by Examples

• Context = input/output examples

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0d 5h 26m</td>
<td>5:00</td>
</tr>
<tr>
<td>0d 4h 57m</td>
<td>4:30</td>
</tr>
<tr>
<td>0d 4h 27m</td>
<td>4:00</td>
</tr>
<tr>
<td>0d 3h 57m</td>
<td>3:30</td>
</tr>
</tbody>
</table>
Application: Code Generation from Natural Language

• Context = natural language description

```
/**
 * Internal helper method for natural logarithm function.
 * @param x original argument of the natural logarithm function
 * @param hiPrec extra bits of precision on output (To Be Confirmed)
 * @return log(x)
 */
```
Application: Test Generation

- Context = program under test
- Probability = bug-detection capability

```java
public int add(int a, int b) {
    ...
}
```

Context

```java
public void testAdd() {
    ...
}
```

Program to be generated
Challenges

• How to estimate the probability $P(prog | context)$?

• How to find program $s$ such that $prog \in Prog$ and $P(prog | context)$ is the largest?
Learning to synthesis (L2S)

• A general framework to address program estimation

• Combining four tools
  • **Rewriting rules**: defining a search problem
  • **Constraint solving**: pruning off invalid choices in each step
  • **Machine-learned models**: estimating the probabilities of choices in each step
  • **Search algorithms**: solving the search problem
Example: Condition Completion

• Given a program without a conditional expression, completing the condition

```java
public static long fibonacci(int n) {
    if ( ?? ) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

• Useful in program repair
  • Many bugs are caused by incorrect conditions
  • Existing work could localize the faulty condition
  • Can we generate a correct condition to replace the incorrect one?

E → E “>12”
  | E “>0”
  | E “+” E
  | “hours”
  | “value”
  | ...

Space of Conditions
Challenge 1: Estimating the Probability

• Idea: Using machine learning
  • To train over a set of programs and their contexts

• Problem: machine learning usually works for classification problems
  • where the number of classes are usually small

• Idea: turn the generation problem into a set of classification problem along the grammar
Decomposing Generation

• In each step, we estimate the probabilities of the rules to expand the left-most non-terminal
  • A classification problem
Probability of the program

$P(prog \mid context) = \prod_i P(rule_i \mid context, prog_i, position_i)$

- $context$: The context of the program
- $prog_i$: The AST generated at the $i$th step
- $position_i$: The non-terminal to be expanded at the $i$th step
- $rule$: the chosen rule at the $i$th step
- $prog$: the complete program

```java
...; if(E > 12) throw new ArgException();
```
Training models

• Train a model for each non-terminal
  • to classify rules expanding this non-terminal

• Training set preparation
  • The original training set:
    • A set of programs
    • Their contexts
  • Decomposing the training set:
    • Parse the programs
    • Extract the rules chosen for each non-terminal
Feature Engineering

• Extract features from
  • context : The context
  • prog\_i : The generated partial AST
  • position\_i : The position of the node to be expanded
Can we choose non-leftmost nonterminal?

- If expanding V gives us more confidence, can we expand V first?

- Yes. We still have

\[ P(\text{prog} \mid \text{context}) = \prod_i P(\text{rule}_i \mid \text{context, prog}_i, \text{position}_i) \]
Can we use a different expansion order?

• Top-down

• Bottom-up

The order may greatly affect the performance of L2S.
Annotations

• Introduce annotations to symbols
  • $E^D$ indicates $E$ can be expanded downward
  • $E^U$ indicates $E$ can be expanded upward
  • $E^{UD}$ indicates $E$ can be expanded in both directions
## From Grammar to Rewriting Rules

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Top-down Rules</th>
<th>Bottom-up Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E “+” E</td>
<td>E^D → E → E^D “+” E^D</td>
<td>E^U → E^U → E “+” E^D</td>
</tr>
<tr>
<td>E → E “&gt;12”</td>
<td>E^D → E → E^D “&gt;12”</td>
<td>E^U → E^U → E “&gt;12”</td>
</tr>
<tr>
<td>E → “hours”</td>
<td>E^D → E → “hours”</td>
<td>“hours”^U → E^U → “hours”</td>
</tr>
</tbody>
</table>

### Creation Rules

- ⇒ E^D  // starting from the root
- ⇒ E^DU  // starting from a middle node
- ⇒ “hours”^U  // starting from a leaf

### Ending Rule

- E^U ⇒ E
Example

• Top-down

⇒ $E^D$

$E^D \Rightarrow E \rightarrow E^D "$>12"

$E^D \rightarrow >12$

$E \rightarrow "$hours"

• Bottom-up

⇒ "$hours"$^U$

$E \rightarrow "$hours"

"hours"$^D \Rightarrow E \rightarrow "$hours"

$E^U \rightarrow E "$+" $E^D$

$E^U \Rightarrow E^U$

$E^U \rightarrow E >12$

$E \rightarrow "$hours"

$E \rightarrow "$hours"
Unambiguity

• A set of rewriting rules are unambiguous if
  • there is at most one unique set of rule applications to construct any program.

• When the rule set is unambiguous, we still have
  • \( P(prog \mid context) = \prod_i P(rule_i \mid context, prog_i, position_i) \)
Challenge 2: How to find the most probable program?

- Local Optimal ≠ Global Optimal

<table>
<thead>
<tr>
<th>E₀</th>
<th>E → E “ &gt; 12” 0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E → E “ &gt; 0” 0.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E₁</th>
<th>E → “hours” 0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E → “value” 0.2</td>
</tr>
<tr>
<td></td>
<td>E → E “+” E 0.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E₂</th>
<th>E → “hours” 0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E → “value” 0.1</td>
</tr>
<tr>
<td></td>
<td>E → E “+” E 0.05</td>
</tr>
</tbody>
</table>

- $0.6 \times 0.2 = 0.12$
- $0.3 \times 0.8 = 0.24$
Idea 1: Use Metaheuristic Search

• Beam Search:
  • Keep $n$ most probable partial programs
  • Expand the programs to get new programs

• Genetic Search:
  • Keep $n$ most probably complete programs
  • Mutate the programs to get new programs
Idea 2: Pruning off Invalid Choices

- Generating constraints from the partial AST
  - Type constraints
  - Size constraints
  - Semantic constraints from E
- Use a solver to determine invalid choices

\[
E^D \Rightarrow E \rightarrow E^D "+" E^D \\
| \quad E \rightarrow E^D ">12" \\
| \quad E \rightarrow "hours"
\]
Summary

• L2S Combines four tools
  • **Rewriting rules**: defining a search problem
  • **Constraint solving**: pruning off invalid choices in each step
  • **Machine-learned models**: estimating the probabilities of choices in each step
  • **Search algorithms**: solving the search problem
Evaluation

• Evaluation 1:
  • Repairing Conditional Expressions

• Evaluation 2:
  • Generating Code from Natural Language Expression
Repairing Conditional Expressions

• Condition bugs are common

hours = convert(value);
+ if (hours > 12)
+   throw new ArithmeticException();

- if (hours >= 24)
  + if (hours > 24)
    withinOneDay=true;

• Steps:
  1. Localize a buggy if condition with SBFL and predicate switching
  2. Synthesize an if condition to replace the buggy one
  3. Validate the new program with tests

  Missing boundary checks
  Conditions too weak or too strong
L2S Configuration

• Rewriting rules
  • Bottom-up
  • Estimate the leftmost variable first

• Machine learning
  • Xgboost
  • Manually designed features

• Constraints
  • Type constraints & size constraints

• Search algorithm
  • Beam search
Results

Benchmark: Defects4J

Number of Repaired Bugs

Precision

Also repaired 8 unique bugs that have never been repaired by any approach.
Generating Code from Natural Language Expression

• Can we generate code automatically to avoid repetitive coding?
• Existing approaches use RNN to translate natural language descriptions to programs
  • Long dependency problem: work poorly on long programs
L2S Configuration

• Rewriting rules
  • Top-down

• Machine learning
  • A CNN-based network

• Constraints
  • Size constraints

• Search algorithm
  • Beam search
A CNN-based Network Architecture
Results

Benchmark: HearthStone

<table>
<thead>
<tr>
<th>Model</th>
<th>StrAcc</th>
<th>Acc+</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPN (Ling et al. 2016)</td>
<td>6.1</td>
<td>–</td>
<td>67.1</td>
</tr>
<tr>
<td>SEQ2TREE (Dong and Lapata 2016)</td>
<td>1.5</td>
<td>–</td>
<td>53.4</td>
</tr>
<tr>
<td>SNM (Yin and Neubig 2017)</td>
<td>16.2</td>
<td>~18.2</td>
<td>75.8</td>
</tr>
<tr>
<td>ASN (Rabinovich, Stern, and Klein 2017)</td>
<td>18.2</td>
<td>–</td>
<td>77.6</td>
</tr>
<tr>
<td>ASN+SUPATT (Rabinovich, Stern, and Klein 2017)</td>
<td>22.7</td>
<td>–</td>
<td>79.2</td>
</tr>
<tr>
<td>Our system</td>
<td>27.3</td>
<td>30.3</td>
<td>79.6</td>
</tr>
</tbody>
</table>
Newest Results

- Replacing CNN with Transformer
  - Transformer: a new neural architecture at 2017
  - The flexibility of L2S allows to easily utilize new models

<table>
<thead>
<tr>
<th>Model</th>
<th>StrAcc</th>
<th>Acc+</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPN (Ling et al., 2016)</td>
<td>6.1</td>
<td>–</td>
<td>67.1</td>
</tr>
<tr>
<td>SEQ2TREE (Dong and Lapata, 2016)</td>
<td>1.5</td>
<td>–</td>
<td>53.4</td>
</tr>
<tr>
<td>YN17 (Yin and Neubig, 2017)</td>
<td>16.2</td>
<td>~18.2</td>
<td>75.8</td>
</tr>
<tr>
<td>ASN (Rabinovich et al., 2017)</td>
<td>18.2</td>
<td>–</td>
<td>77.6</td>
</tr>
<tr>
<td>ReCode (Hayati et al., 2018)</td>
<td>19.6</td>
<td>–</td>
<td>78.4</td>
</tr>
<tr>
<td>CodeTrans-A</td>
<td>25.8</td>
<td>25.8</td>
<td>79.3</td>
</tr>
<tr>
<td>Structured</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASN+SUPATT (Rabinovich et al., 2017)</td>
<td>22.7</td>
<td>–</td>
<td>79.2</td>
</tr>
<tr>
<td>SZM19 (Sun et al., 2019)</td>
<td>27.3</td>
<td>30.3</td>
<td>79.6</td>
</tr>
<tr>
<td><strong>CodeTrans-B</strong></td>
<td><strong>31.8</strong></td>
<td><strong>33.3</strong></td>
<td><strong>80.8</strong></td>
</tr>
</tbody>
</table>
Conclusion

• Program Estimation: to find the most probable program under a context
• L2S: combining four tools to solve program estimation
• Why worked?
  • Machine learning to estimate probability
  • Rewriting rules and constraints to confine the space
  • Search algorithms to locate the best program
• Better to combine the tools we have
## 深度学习程序缺陷实证研究

<table>
<thead>
<tr>
<th>现象</th>
<th>原因</th>
<th>挑战</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 崩溃/异常(64%)</td>
<td>• 错误结构或参数(22%)</td>
<td>• 概率正确</td>
</tr>
<tr>
<td>• 效果差(23%)</td>
<td>• 张量维数不匹配(14%)</td>
<td>• 巧合正确</td>
</tr>
<tr>
<td>• 效率低(5%)</td>
<td>• 混淆Tensorflow和传统语言(10%)</td>
<td>• 执行随机</td>
</tr>
</tbody>
</table>

- • Tensorflow API升级(25%)
- • 误用Tensorflow API(19%)
- • 网络模型低效(1.7%)

- • 混淆
- • Tensorflow
- • 网络模型低效
- • 网络模型低效
Thank you for listening!

Main References:

