Learning to Synthesize

Yingfei Xiong
Peking University
IBF 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)

Low-Quality Patches

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

How to deal with the weak specification?

• Find the most-likely patch under the current context

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

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

[1] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, Lu Zhang. Precise Condition Synthesis for Program Repair. ICSE’17.
Program Estimation

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

• We define this problem as **program estimation**:
  • Given a context \(c\), a (weak) specification \(E\), and a space of programs \(S\),
    find program \(s = \arg\max_{s \in S \land E(s)} P(s \mid c)\)?

• A sub-problem of program synthesis

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

```java
... Math.abs(n) < 1
n == Integer.Max_VALUE
n < 19
n < 21
```
Application: Test-based Program Repair

- Context = buggy program & at least one failed test

Passing Test

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(int a, 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) {
    ...
}
public void testAdd() {
    ...
}
```

Context

Program to be generated
Challenges

• How to estimate the probability $P(\text{Prog} \mid \text{Context})$?

• How to find program $s$ such that $E(s)$ and $P(s \mid \text{context})$ 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

Expand E with E -> E “> 12”

Expand E with E -> “hours”
Probability of the program

\[ P(\text{prog} \mid \text{context} ) = \prod_i P(\text{rule}_i \mid \text{context}, \text{prog}_i, \text{position}_i) \]

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

...; if( \text{E} \) throw new \\
| ArgException(); | \text{context} | \text{prog}_i | \text{context} | \text{position}_i |
|--------------------|----------------|----------------|--------------------|
|                     | \text{E}         | \text{E}         |                    |
|                     | \text{E}         | \text{E}         | \text{E}           |
|                     | >12              | >12             | \text{E}           |
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
  • \textit{context} : The context
  • \textit{prog}_i : The generated partial AST
  • \textit{position}_i : The position of the node to be expanded

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

\textit{context} \quad \textit{prog}_i \quad \textit{position}_i \quad \textit{context}
Can we choose non-leftmost nonterminal?

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

• Yes. We still have

\[
P(prog | context) = \prod_i P(rule_i | context, prog_i, position_i)
\]
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>$E \rightarrow E \text{ &quot;+&quot; } E$</td>
<td>$E^D \Rightarrow E \rightarrow E^D \text{ &quot;+&quot; } E^D$</td>
<td>$E^U \Rightarrow E^U \rightarrow E \text{ &quot;+&quot; } E^D$</td>
</tr>
<tr>
<td>$E \rightarrow E \text{ &quot;&gt;12&quot;}$</td>
<td>$E^D \Rightarrow E \rightarrow E^D \text{ &quot;&gt;12&quot;}$</td>
<td>$E^U \Rightarrow E^U \rightarrow E \text{ &quot;&gt;12&quot;}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{ &quot;hours&quot;}$</td>
<td>$E^D \Rightarrow E \rightarrow \text{ &quot;hours&quot;}$</td>
<td>\text{ &quot;hours&quot;}^U \Rightarrow E^U \rightarrow \text{ &quot;hours&quot;}$</td>
</tr>
</tbody>
</table>

**Creation Rules**

- $\Rightarrow E^D$  // starting from the root
- $\Rightarrow E^{DU}$  // starting from a middle node
- $\Rightarrow \text{ "hours"}^U$  // starting from a leaf

**Ending Rule**

$E^U \Rightarrow E$
Example

• Top-down

⇒ $E^D$

⇒ $E ightarrow E^D \text{ "} >12\text{"}$

$E^D \Rightarrow E$ >12

$E \rightarrow \text{ "} hours\text{"}$

$E \rightarrow >12$

• Bottom-up

⇒ "hours"$^U$

⇒ $E^U \rightarrow \text{ "} hours\text{"}$

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

$E^U \Rightarrow E$

$E \rightarrow >12$

$E \rightarrow \text{ "} hours\text{"}$

$E \rightarrow >12$
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
  • \( P(prog | context) = \prod_i P(rule_i | context, prog_i, position_i) \)
Challenge 2: How to find the most probable program?

• Local Optimal ≠ Global Optimal

\[
\begin{align*}
E_0 & : E \rightarrow E \text{“} > 12\text{”} 0.3 \\
& : E \rightarrow E \text{“} > 0\text{”} 0.6 \\
E_1 & : E \rightarrow \text{“}hours\text{”} 0.1 \\
& : E \rightarrow \text{“}value\text{”} 0.2 \\
& : E \rightarrow E \text{“} + \text{”} E 0.05 \\
E_2 & : E \rightarrow \text{“}hours\text{”} 0.8 \\
& : E \rightarrow \text{“}value\text{”} 0.1 \\
& : E \rightarrow E \text{“} + \text{”} E 0.05
\end{align*}
\]

\[
\begin{align*}
0.6 \times 0.2 & = 0.12 \\
0.3 \times 0.8 & = 0.24
\end{align*}
\]
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

\[
E^D \Rightarrow E \rightarrow E^D \text{ "+" } E^D
\]
\[
| \quad E \rightarrow E^D \text{ ">12"}
\]
\[
| \quad E \rightarrow \text{ "hours"}
\]
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

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

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

Missing boundary checks

Conditions too weak or too strong

• 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
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

- ConCap
- SimFix
- ACS
- Elixir

- IF-Related
- General

Precision

- ConCap
- SimFix
- ACS
- Elixir

- IF-Related
- General

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><strong>Our system</strong></td>
<td><strong>27.3</strong></td>
<td><strong>30.3</strong></td>
<td><strong>79.6</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
Thank you for listening!

Main References: