CodeHint: Dynamic and Interactive Synthesis of Code Snippets

Joel Galenson, Ras Bodik, Koushik Sen
UC Berkeley
Motivation

Specification

Test cases

Desired type
CodeHint: Autocomplete for the modern age

- Autocomplete is useful but very limited.
- Our improvements:
  - Being dynamic
  - General specifications
  - Synthesis
Overview

Source code → Find locals → Evaluate → Candidates → Refine

Test case → Find locals

Specification → CodeHint

Combine → Candidates
Example: The problem

```java
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        int x = e.getX(), y = e.getY();
        // Get the menu bar.
    }
});
```

![Diagram of a file structure]

- File
  - home
    - Alice
    - Bob
    - Eve
Example: First step

```
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        int x = e.getX(), y = e.getY();
        // Get the menu bar.
    }
});
```

“Find a JMenuBar”

```
rv instanceof JMenuBar
```
Demo
Basic algorithm

- Find local variables, do BFS over Java statements, show user those that pass spec.
  - Actually evaluate these statements, including file I/O, reflection, etc.

```
Iteration 1
x
y

Iteration 2
+  
  x  
  y

max
  x  
  y

Iteration 3
average
  max
    x  
    y

+  
  x  
  y

rv instanceof JMenuBar
```

rv instanceof JMenuBar
First iteration

• Find local variables.

```java
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        int x = e.getX(), y = e.getY();
        // Get the menu bar.
    }
});
```

x   y  tree  this  null  ...
Second iteration

- Get each expression's type and combine it with others in type-correct ways.
Side effects

- Handle in-memory and external effects.
Equivalence classes

- Group equivalent code to avoid unneeded work.

Next iteration:

End of search:
Probabilistic model

• Mined 10MLOC to build model of likely code.

\[
P(m) = P(m|T)P(T) = \frac{\text{# accesses of } m \text{ on } T}{\text{# of accesses on } T} \times \frac{\text{# of accesses on } T}{\text{# of accesses}} = \frac{\text{# accesses of } m \text{ on } T}{\text{# of accesses}}
\]

Probability of using constant \( c \) as argument \( i \) to method \( m \):

\[
P(c, m, i) = P(c, m, i|c)P(c) = \frac{\text{# uses of } c \text{ on method } m \text{ at index } i}{\text{# of uses of } c} \times \frac{\text{# of uses of } c}{\text{# of uses}} = \frac{\text{# uses of } c \text{ on method } m \text{ at index } i}{\text{# of uses}}
\]

• Use to avoid unlikely expressions.
Third iteration and result

Third iteration:

rv instanceof JMenuBar

Result:

rv instanceof JMenuBar
Refinement

• Users can give another demonstration in a different state to refine the results.
Synthesis from specifications

Context-dependence

Context-independent

API exploration

Static type
rv instanceof JMenuBar

Program synthesis

Correctness condition
isSorted(array)

x.toString().contains("Eve")

Programming by demonstration

Value demonstrations
x == 42
## Empirical results

In real-world code, ~95% of expressions need \( \leq 3 \) iterations and ~99% need \( \leq 4 \).

### Normal algorithm

<table>
<thead>
<tr>
<th>Iteration 2</th>
<th>Iteration 3</th>
<th>Iteration 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Time</td>
<td>#</td>
</tr>
<tr>
<td>P 1</td>
<td>34</td>
<td>0.1</td>
</tr>
<tr>
<td>P 2</td>
<td>52</td>
<td>0.1</td>
</tr>
<tr>
<td>P 3</td>
<td>53</td>
<td>0.1</td>
</tr>
<tr>
<td>P 4</td>
<td>7</td>
<td>0.1</td>
</tr>
<tr>
<td>P 5</td>
<td>22</td>
<td>0.1</td>
</tr>
<tr>
<td>S 1</td>
<td>8</td>
<td>0.2</td>
</tr>
<tr>
<td>S 2</td>
<td>12</td>
<td>0.1</td>
</tr>
<tr>
<td>S 3</td>
<td>70</td>
<td>1.0</td>
</tr>
<tr>
<td>S 4</td>
<td>103</td>
<td>0.3</td>
</tr>
<tr>
<td>S 5</td>
<td>32</td>
<td>0.2</td>
</tr>
<tr>
<td>R 1</td>
<td>20</td>
<td>0.1</td>
</tr>
<tr>
<td>R 2</td>
<td>12</td>
<td>0.0</td>
</tr>
<tr>
<td>R 3</td>
<td>8</td>
<td>0.2</td>
</tr>
<tr>
<td>R 4</td>
<td>6</td>
<td>0.0</td>
</tr>
<tr>
<td>R 5</td>
<td>24</td>
<td>0.2</td>
</tr>
<tr>
<td>Avg</td>
<td>30.9</td>
<td>0.2</td>
</tr>
<tr>
<td>Med</td>
<td>22</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Our algorithms work well in practice.
User studies

- Completed two user studies with 28 subjects.
- Found statistically-significant productivity improvements:
  - Fewer bugs and more tasks completed in less time.

CodeHint makes programmers more productive.
Future work

- Improving the probabilistic model
- CodeHint for JavaScript
  - https://github.com/jgalenson/codehint.js
- Integrating symbolic techniques
Probabilistic search

ggetWindowAncestor

SwingUtilities tree

; ;

foo bar

executor.shutdown();
try {
    while (!executor.isTerminated())
        Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
Summary

- Dynamic and interactive synthesis
- Autocomplete for the modern age
- User studies showed productivity improvements

Thanks!
https://jgalenson.github.io/codehint/