CodeHint: Dynamic and Interactive Synthesis of Code Snippets

Joel Galenson, Philip Reames, Ratislav Bodik, Björn Hartmann, Koushik Sen
Research Questions
Motivating Research Questions

1. How can users find code snippets using whatever partial information they have about the desired result?
Motivating Research Questions

1. How can users find code snippets using whatever partial information they have about the desired result?

2. Can a dynamic approach to code generation and completion be more useful than existing static methods?
Motivating Research Questions

1. How can users find code snippets using whatever partial information they have about the desired result?

2. Can a dynamic approach to code generation and completion be more useful than existing static methods?

3. How can the search procedure for candidate statements be improved?
Contributions
Contributions

A method for synthesizing code using:

Dynamic Analysis

Intuitive Specification Language

Interactive User Input
Contributions

An algorithm that can generate relevant code snippets based on user constraints and probabilistic model

Powerful enough to handle I/O, reflections, and native calls in the host language
Contributions

An implementation of preceding ideas in the form of an Eclipse plug-in, for Java code

Some empirical studies on the implementation’s effectiveness in real-world scenarios
Key Ideas
Key Ideas

- Programmer expresses their intuition about the result and the IDE synthesizes code fragments.
- Most tools before CodeHint that help programmers find code fragments, rely on static information. They are inexpressive.
- Use Dynamic Analysis!
- Why is dynamic analysis better than static?
Key Ideas

● Take advantage of dynamic context information

● Example:
  Dereferencing exactly the expressions that do not evaluate to **NULL** in the current context
Key Ideas

The Specification:

Programmer expresses their partial knowledge about the result using predicates called partial dynamic specification (PDSpec)

- Pdspecs can be a constraint on the desired value, type or any other property.
- example: x instanceof MenuBar
Algorithm

Given this specification, CodeHint will begin an iterative search for expressions that satisfy the pdspec.

- **First Iteration:**
- CodeHint queries the debugger, searches local variables and special values like this, null.
Algorithm

Second Iteration:

● CodeHint combines simple expressions into complicated ones according to the language grammar
● All accessible methods available on a type are queried.
● Try all combinations.
● Evaluations might have side effects, so CodeHint has to keep undoing them
● Uses Java's security manager to disable external side effects like deleting files.
● Equivalent expressions will be grouped to avoid duplication.
Algorithm

Third Iteration:

- Over 10 million Java LOC analysed, and a probabilistic model is developed that helps guide the search.
- The probabilistic model will guide the search towards the most likely ones.
Key Ideas

- User can give extra hints using **Skeletons**.

- Skeletons are normal code with holes representing unknowns.

- **Example:** `MyObject.myMethod(??)`

- `?? = missing portion`
Explanatory Example
Objective: To remove ALL integers of a specified value, from a list.

```java
void RemoveAllFromList(
    List<Integer> ls,
    int x)
{
    // Code Required!
}
```
Objective: To remove ALL integers of a specified value, from a list.

```java
void RemoveAllFromList(
    List<Integer> ls,
    int x)
{
    // Code Required!
}
```

Sample Input

```
ls = [0, 1, 2]
x = 0
```

PDSpec

```
!ls.contains((Integer)0)
```
Objective: To remove ALL integers of a specified value, from a list.

```java
void RemoveAllFromList(
    List<Integer> ls,
    int x)
{
    // Code Required!
}
```

**Sample Input**

- `ls = [0, 1, 2]`
- `x = 0`

**PDSpec**

- `!ls.contains((Integer)0)`

**Candidates**

- `ls.remove(0)`
- `ls.removeAll((Integer)x)`
- `ls.remove((Integer)x)`
- `ls.removeAll((Integer)0)`
- `ls.clear()`
Objective: To remove ALL integers of a specified value, from a list.

```java
void RemoveAllFromList(
    List<Integer> ls,
    int x)
{
    // Code Required!
}
```

Sample Input: 
```java
ls = [2 , 3 , 4 , 3]
x = 3
```

PDSpec: 
```java
!ls.contains((Integer)3)
```
Objective: To remove ALL integers of a specified value, from a list.

```java
void RemoveAllFromList(
    List<Integer> ls,
    int x)
{
    // Code Required!
}
```

**Candidates**

- `ls.remove(0)`
- `ls.removeAll((Integer)x)`
- `ls.remove((Integer)x)`
- `ls.removeAll((Integer)0)`
- `ls.clear()`

**Sample Input**

- `ls = [2, 3, 4, 3]`
- `x = 3`

**PDSpec**

- `!ls.contains((Integer)3)`
Evaluations
User Evaluation

I.e. *How useful is it in practice?*

- Ability to complete task?
- Task completion time?
- Quality of code?

Tested using 28 people, divided into two groups which worked on same tasks, independent of each other. One group used CodeHint, other group did not.
User Evaluation

Results

<table>
<thead>
<tr>
<th></th>
<th>Without CodeHint</th>
<th>With CodeHint</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Success Rate</strong></td>
<td>27%</td>
<td>69%</td>
</tr>
<tr>
<td><strong>Completion Time</strong></td>
<td>92 s</td>
<td>46 s</td>
</tr>
<tr>
<td><strong>Number of bugs</strong></td>
<td>24</td>
<td>11</td>
</tr>
</tbody>
</table>

Statistically Significant Results!
CodeHint helps!
Performance Evaluation

I.e. *Is the tool efficient?*

Search time?

Is probabilistic model advantageous?
Performance Evaluation

Time needed to search & evaluate till various depths

<table>
<thead>
<tr>
<th>Depth</th>
<th>Average</th>
<th>Depth</th>
<th>Average</th>
<th>Depth</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth = 2</td>
<td>0.5 s</td>
<td>Depth = 3</td>
<td>1.3 s</td>
<td>Depth = 4</td>
<td>5.3 s</td>
</tr>
<tr>
<td>Median</td>
<td>0.4 s</td>
<td>Median</td>
<td>1.1 s</td>
<td>Median</td>
<td>3.6 s</td>
</tr>
</tbody>
</table>
Performance Evaluation

Advantage of Probabilistic Model + Heuristics
(Measured by number of expressions evaluated till depth = 3)

<table>
<thead>
<tr>
<th></th>
<th>Standard</th>
<th>Without Heuristics</th>
<th>Brute Force</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>412.9</td>
<td>53769.2</td>
<td>44857654.2</td>
</tr>
<tr>
<td>Median</td>
<td>234</td>
<td>4457</td>
<td>115410</td>
</tr>
</tbody>
</table>

CodeHint is efficient!
The probabilistic model helps!
Discussion Questions
Discussion Questions

Is it better to break up a CodeHint request into multiple intermediate steps, or to chain method calls into one single statement?
Discussion Questions

The developers used twenty-eight users to evaluate CodeHint’s effectiveness. Is this enough to achieve confidence in the result?
Discussion Questions

This implementation was done for Java. How could CodeHint’s methods work for other programming languages?
Discussion Questions

If a working codebase were used to train the probabilistic model instead of the original ten million line codebase, how could CodeHint’s functionality be affected?
Discussion Questions

How could CodeHint be used for debugging?