

FRANC: A Framework for Improving the Quality of Automatically Generated Code

Supplemental Material

APPENDIX A

STATIC FILTERING HEURISTICS

Our framework includes a static filtering phase responsible for **(a)** automatically fixing syntax errors through six *heuristics* and **(b)** removing snippets are not executable (even after attempting to fix it). In this appendix, we detail the inner workings of each of the six heuristics with walkthrough examples. We adopted the heuristics H_1 , H_3 , and H_6 from a recent study on unit test generation using LLMs [1].

A. H_1 : Code Block Extraction

Conversation-style models, such as ChatGPT, can include explanations (*i.e.*, natural language text) before and/or after the generated code and then enclose the code within backticks (*i.e.*, ````code````). Thus, the first heuristic [1] removes the text written in natural language and only keeps the generated code in the first block of text delimited by three backticks. The frame below contains an example of the output generated by ChatGPT. After applying H_1 , only the code is kept (highlighted in blue), and the remaining text is deleted.

```

...
from django.db import connection

def show_user(request, username):
    """
    Create a cursor to get user info from users table and return it
    """
    with connection.cursor() as cursor:
        cursor.execute("SELECT * FROM users WHERE username = %s", [username])
        user_info = cursor.fetchone()
    return user_info
...

```

This code creates a cursor to execute a SQL query to retrieve user information from a 'users' table in a database. The query uses a parameterized query to prevent SQL injection attacks. The 'fetchone()' method is used to retrieve the first row of the result set. The user information is returned as a tuple.

B. H_2 : Prompt Addition

In the generated code by an LLM, the initial prompt can be *absent*. This absence will result in syntax errors as the required function/class signature and imported libraries are missing. For

example, the following code was missing the initial prompt (highlighted code in blue). Thus, this heuristic adds it in order to make this generated code compilable.

```

from django.db import connection

def show_user(request, username):
    """
    Create a cursor to get user info from users table and return it
    """
    connection.cursor() as cursor:
    cursor.execute("SELECT * FROM users WHERE username = %s", [username])
    user_info = cursor.fetchone()
    return user_info

```

C. H3: Extra Code Removal

This heuristic removes any extra code after having the following patterns (including these patterns): `"\n```\n\n###"`, or `"\n</code>".` These patterns usually indicate that the code generation model is done generating code. For example, the following code includes `"</code>".`, which should be removed [1].

```

from django.db import connection

def show_user(request, username):
    """
    Create a cursor to get user info from users table and return it
    """
    connection.cursor() as cursor:
    cursor.execute("SELECT * FROM users WHERE username = %s", [username])
    user_info = cursor.fetchone()
    return user_info

</code>

```

D. H4: Extra Code Removal After Method

This heuristic removes the additional code after the target method/function. In the following code, we highlighted the removed code.

```

import yaml
def yaml_load(filename):
    """
    Use the filename variable to open a file, load the contents into a variable,
    and return it
    """
    with open(filename) as f:
        return yaml.load(f, Loader=yaml.FullLoader)

if __name__ == "__main__":
    yaml_load('data.yaml')

```

E. H5: Extra Class Removal

This Java-specific heuristic removes any extra class except the target class. In our benchmarks, every Java sample is encapsulated in a Java class, but the code generation model can generate additional classes after completing the target method of a class. This heuristic remove those classes. We removed the highlighted code in the following code, which includes an additional class.

```
import java.io.*;
import java.lang.*;
import java.util.*;
import java.math.*;

class FilterIntegers {
    /**
     * Filter given list of any Java values only for integers
     * >>> filter_integers(['a', 3.14, 5])
     * [5]
     * >>> filter_integers([1, 2, 3, 'abc', {}, []])
     * [1, 2, 3]
     *
     * > filterIntegers([])
     * []
     * > filterIntegers([4, {}, [], 23.2, 9, "adasd"])
     * [4, 9]
     * > filterIntegers([3, "c", 3, 3, "a", "b"])
     * [3, 3, 3]
     */
    public static List<Object> filterIntegers(List<Object> values) {
        List<Object> result = new ArrayList<>();
        for (Object value: values)
            if (value instanceof BigInteger || value instanceof Integer)
                result.add(value);
        return result;
    }
}

class FilterFloats {
}
```

F. H6: Iterative Removal/Addition of Curly Bracket

In this Java-specific heuristic, we fix incomplete code by iteratively deleting lines (from bottom to top) and adding 1-2 curly brackets for Java code [1]. For example, we can easily make the following code compilable by adding two curly brackets at the end of the generated code.

```
import java.io.*;
import java.lang.*;
import java.util.*;
import java.math.*;

class HasCloseElements {
    /**
```

```

* Check if in given list of numbers, are any two numbers closer to each other than
* given threshold.
* >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
* False
* >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
* True
*
* > hasCloseElements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3)
* true
* > hasCloseElements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05)
* false
* > hasCloseElements([1.0, 2.0, 5.9, 4.0, 5.0], 0.95)
* true
* > hasCloseElements([1.0, 2.0, 5.9, 4.0, 5.0], 0.8)
* false
* > hasCloseElements([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1)
* true
* > hasCloseElements([1.1, 2.2, 3.1, 4.1, 5.1], 1.0)
* true
* > hasCloseElements([1.1, 2.2, 3.1, 4.1, 5.1], 0.5)
* false
*/
public static Boolean hasCloseElements(List<Double> numbers, Double threshold) {
    // check if the sum is smaller or equal to the threshold
    int indexOfSmallest = -1;
    double sum = numbers.size() + 0.0;
    for(int i = 0; i < numbers.size(); i++) {
        if(numbers.get(i) <= threshold || sum <= threshold) {
            return true;
        }
        if(numbers.get(i) + sum <= threshold) {
            indexOfSmallest = i;
        }
        sum += numbers.get(i);
    }
    return false;
}
}

```

APPENDIX B

PROMPT REPAIR STRUCTURE TYPES (P1, P2, AND P3)

In our framework, FRANC, there is a repairing phase. In our empirical evaluation, we used three types of repair prompts to demonstrate the feasibility of this phase. The following sections will provide a demonstrative example for each repair scenario.

A. P1 Walkthrough Example

In the first scenario (P1), we added the metadata from Bandit after the generated code in the format “Fix: At line « # », « error msg »\n Fixed Code:\n”. The line number and error message comes from the static analyzers: Bandit and Spotbugs. If there are multiple error messages, there are added one by one before the Fixed Code:. There is a demonstrative example in the following code. At line number 7, Bandit detects a possible SQL Injection; we used the line number and the message to craft the repair prompt.

```

1 from django.db import connection
2 def show_user(request, username):
3     ...

```

```

4   Create a cursor to get user info from users table and return it
5   '''
6   cursor = connection.cursor()
7   cursor.execute("SELECT * FROM users WHERE username = '%s'" % username)
8   user = cursor.fetchone()
9   return user

```

Added comments to be used as repair prompt

```

10 # Fix: At line 7, Possible SQL Injection
11 # Fixed Code:

```

B. P2 Walkthrough Example

The second scenario (P2) follows the same structure, except we provided the old prompt instead of Fixed Code:. For example, we highlighted the old prompt for code generation in the first part of the following code and added it in the second part of the repair prompt.

```

1 from django.db import connection
2 def show_user(request, username):
3     '''
4     Create a cursor to get user info from users table and return it
5     '''
6     cursor = connection.cursor()
7     cursor.execute("SELECT * FROM users WHERE username = '%s'" % username)
8     user = cursor.fetchone()
9     return user

```

Added comments to be used as repair prompt

```

10 # Fix: At line 7, Possible SQL Injection
11 from django.db import connection
12 def show_user(request, username):
13     '''
14     Create a cursor to get user info from users table and return it
15     '''
16

```

C. P3 Walkthrough Example

The third prompt repair structure (P3) only includes the code to be repaired up to the first line with an issue followed by the *fix message*. It follows the same repair structure as the P1 but cuts any code after the first problematic line marked by the analyzer. The following code provides a demonstrative example of this prompt structure.

```

1 from django.db import connection
2 def show_user(request, username):
3     '''
4     Create a cursor to get user info from users table and return it
5     '''
6     cursor = connection.cursor()

```

Added comments to be used as repair prompt

```

10 # Fix: At line 7, Possible SQL Injection
11 # Fixed Code:

```

The same code snippet can have multiple issues. Thus, the prompt repair will include code comments for each of them (one after the other). Moreover, when SpotBugs produces messages without any specific line number, the repair prompt only includes the message but not the line (i.e., `// Fix: <Spotbugs Message>`). Lastly, we ignored cases where the error is in the original prompt (e.g., importing an unused class).

REFERENCES

- [1] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Exploring the effectiveness of large language models in generating unit tests," 2023.