Correlating Program Code to Output for Supporting Program Understanding

Miyu Satoh, Seikoh Nishita,

***Abstract*—Program understanding is important for novice programers and advanced programers to comprehend what will happen in execution and why their programs give wrong output. This paper describes a technique to make code/output correspondence in runtime of a given program. Then, we show how this technique is useful for program understanding. In addition, we report a development of a turtle graphics editor as an application of our technique.**

***Index Terms*—program understanding, program visualization, dynamic program analysis.**

I. Introduction

PROGRAM understanding in this paper means the skills of tracing execution and explanation of program behavior; both skills are based on knowledge of the language syntax and the computational model. Program understanding is important for novice programmers, because they need to read and understand sample program code in textbooks as a previous step of writing codes. Program understanding is also important for advanced programmers. Because they often tackle large scale program codes with various libraries, they need to modularize codes, and to understand roles of each module clearly.

For program understanding, program visualization techniques have been proposed in context of the program education and debugging. These techniques show a program code with its execution state including values in variables, call trees and so on. During the visualization, each statement in the program code looks like an actor on the execution state.

This paper describes a technique to correlate program code with its output to support program understanding. Our technique assumes output is a collection of parts like characters, bytes, components of GUI and fragments of figures. When a statement dumps parts of output in runtime, our technique makes information of correspondence between the statement and the parts. This information is leveraged to support program understanding. This paper also shows how this technique can be used for program understanding. In addition, this paper reports a development of a program editor for turtle graphics that correlate program code to fragments of figures in turtle graphics.

Section II introduces related works on program visualization for program understanding. Section III describes our technique and its usefulness. Section IV reports the development of the program editor. Section V discusses the technique and future works, and section VI concludes.

Manuscript received Dec 20, 2018; revised Jan 10, 2019.

M.Satoh and S.Nishita are with Department of Computer Science, Faculty of Engineering, Takushoku University, 815-1, Tatemachi, Hachioji, Tokyo, Japan. e-mail: snishita@cs.takushoku-u.ac.jp.

II. Program Visualization for Program Understanding

This section describes previous works on program visualization for program understanding. All works described here link program code to runtime state.

ETV [1] helps students to understand the behaviors of programs by presenting a graphical representation of the execution trace data. The execution trace data includes arguments, values and the program points in multiple code viewers. When a user selects a node in the call tree, ETV shows the runtime state by the execution trace data. The program point highlighted in ETV indicates the moment at the execution of the runtime state shown.

The Online Python Tutor [2], [3] visualizes objects, variables and stack frames allowing students to inspect the runtime state of their code at every moment of the execution.

To address misconceptions about code, Lieber et al. proposed IDE extension named Theseus that visualizes runtime behavior within a JavaScript code editor [4]. Theseus finds code/behavior correspondence and identifies which code is responsible for a particular program behavior.

Doppio [5] tracks and visualizes UI flows and their changes based on source code. While a developer runs an interactive app, Doppio captures and logs runtime state. Then, it presents the screenshots, video clips, code snippets, UI regions and run-time argument values for each user input like a click event.

Satoh et al. proposed a system to support understanding GUI programs written in Java. [6] The system visualizes reference relationship of classes and thumbnails of the GUI windows. Thumbnails are linked with statements of program codes, that is, thumbnails show how GUI windows are changed by the statements in each moment of runtime.

To reveal runtime behavior during both normal execution and debugging, Hoffswell et al. proposed a techniques to visualize program variables directly within the source code [7]. The technique links variables on the code viewer to their value in runtime. In addition, it supports simple value and

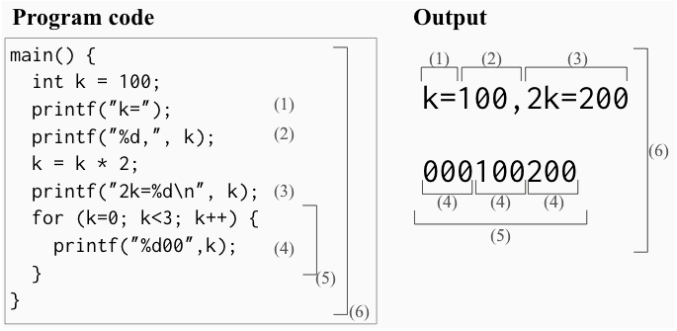

Fig. 1. Correlation of program code to output

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

a set of values, and represents its snapshot and changes in every moment with histograms, line charts and so on.

All of these previous works visualize both codes and the runtime state. Especially, the code is used to indicate the moment in the execution, and it looks like an actor to change the runtime state.

III. Correlation of Program Code to Output

This section proposes a technique to correlate program code with output. We assume that the output is a collection of *parts*. For example, character stream, byte stream, GUI windows and figures are all output, whose parts are characters, bytes, GUI components and fragments of the figure respectively. In addition, we also assume that there is a given program code which runs without any error and dumps an output, i.e. a sequence of parts.

This section first proposes our technique with the rule of the correlation, and secondly illustrates usefulness of the technique by examples of its application.

A. Rule of the Correlation

The correlation is based on the following rules.

- R1: correlate every statement in the given code with parts dumped by the statement in runtime.
- R2: correlate every code block in the given code with all parts dumped by the all statements in the code block in runtime.
- R3: correlate every function definition in the given code with parts dumped by the function calls.

R1 is basis of the rules, and R2 is a natural extension of R1 to a collection of statements. Fig.1 illustrates our technique, where simple C program and its output are given. Blue lines and numbers indicate the correlation information; `printf` statements (1), (2) and (3) are correlated with character sequences (1), (2) and (3) respectively. In contrast, the statement (4) is correlated with three character sequences (4) in the output, because the statement (4) is executed three times by the enclosing `for` statement. On the other hand, the `for` statement(5) and the main block(6) are correlated with multiple sequences. Since the program is nested by blocks, output also makes hierarchical structure by the correlation.

The key perspectives of the correlation is that our technique treats statements in the code and parts in the output as both objects for correlation. In contrast, many of previous works considers that a statement leads some action on the runtime state.

B. Program Visualization with the Correlation

Our technique is available to be applied in program visualization. Fig.2 shows an image of the program visualization tool with our technique; the tool has two panes, a code viewer and an output text area. When a user clicks a statement in the viewer pane, the tool highlights the parts in the output pane, and vice versa. If the statement selected is in loop statement, the tool highlights the parts in one or more colors to show they are iterated parts.

The application of our technique in program visualization has some limitation. While output is simply a one of runtime state in program execution, there are various kind of runtime state. Moreover, output usually follows calculations and data structure manipulations, whose visualization are needed to understand the reason why output is so.

Nevertheless, our technique is useful in program visualization in some cases.

1) Case1: Introductory Loop-Programming Exercise: In conventional hello-world style learning activity, the standard output is always a goal of programming drills. The exercises don't involve complexity of calculations or data structures. Since the code/output correspondence in this case is relatively simple, the program visualization with our technique is useful for novice programmers.

The followings are the advantages of the tool considered:

- To novice programmers, the tool provides opportunities to inspect program behaviors in many cases by diddling codes. Fig.3 is an example of paper-based teaching material. The tool provides the same information to novice programmers. In addition, when a novice programmer

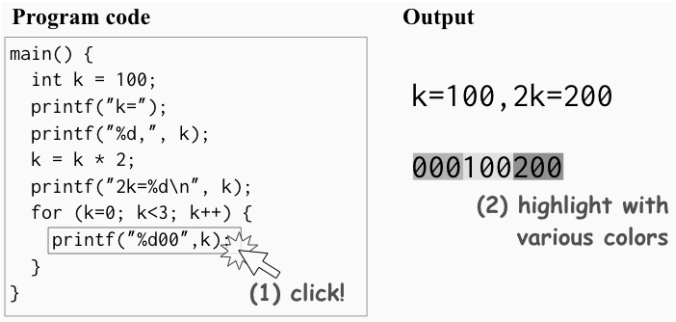

Fig. 2. Highlight parts in output correlated to the selected statement

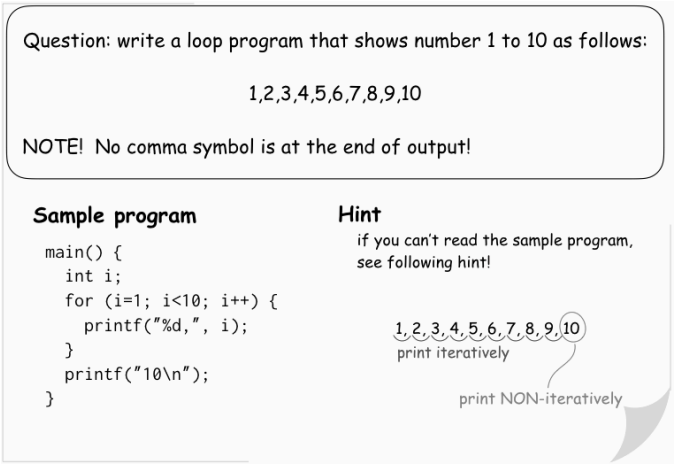

Fig. 3. An example of a handout on loop programming

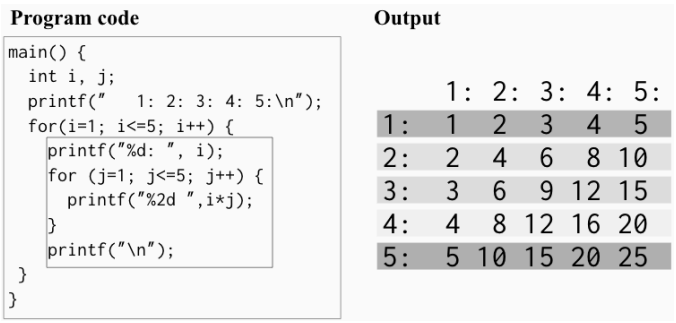

Fig. 4. An example of code to output a part of multiplication table

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

modifies the given code, the tool shows alternative information with the modified code. Diddling codes and inspecting the correlation are more helpful for their program understanding than static contents on paper.

- The tool can demonstrate program behaviors of every code blocks. Fig.4 describes a program code that generates a part of the multiplication table. The program code has a nested `for` statement, and the selected block corresponds to each row of the table, which is illustrated by the tool.

C. Case2: Debugging Programs

Fig.5 shows a snippet of a TypeScript program with React to make a web form, which has the suspicious semi-colon symbols. Which semi-colon in the code causes the symbols on the web page? Debugging in this case, we have to find the code point that dumps the semi-colon symbols. Modern web browsers have the developers mode that illustrate correspondence of the web page and its DOM tree. However, this feature does not work fine in this case, because JSX elements in React are actually functions to dump HTML tags, and there is non-ignorable gap between DOM tree and JSX elements.

Similar problem exists at program codes with modern libraries that has a purpose like code simplicity or security-awareness.

We expect that our technique would support to debug in such situation, because it makes the code/output correspondence. A web developing tool with our technique would provide a user interface to access directly to a bug in the code from unexpected elements on a web page.

IV. Development of Program Editor for Turtle Graphics

The previous section describes a technique to correlate a program code with output. However, realizability or algorithm of our technique is not described yet. As an application of the technique, this section reports a development of a turtle graphics tool, and gives a brief sketch of the correlation algorithm in it.

A. Turtle Graphics Tool with The Correlation

The turtle graphics tool supports novice programers to write and execute turtle graphics programs. It is based on Python turtle library [8]. The editor comprises two components: one is a program editor, and the other is a turtle graphics viewer. A user can click and drag on the editor and the viewer to select a part of the code and the figure.

The turtle graphics tool correlates statements in the editor with fragments of figures in the viewer. When a user selects a statement, the tool highlights fragments which is dumped by the statement. In addition, a statement is also highlighted by user's clicking a fragment in the viewer. Fig.6 shows appearance of the tool; one of the `forward` statements in the editor is selected; this statement dumps three highlighted segments.

Our tool has a feature to expand/shrink selected area in the editor and the viewer. In the expansion process, a selected area of single statement is expanded to one of a code block including the statement. On the other hand, an area of a code block is expanded to a single control statement that has the code block. Fig.7 illustrates how the expansion/shrinkage proceeds: the first selected statement is the `forward` statement(1), and the expansion proceeds the block(2) and the `for` statement(3) in this order. On the other hand, the shrinkage proceeds in reverse.

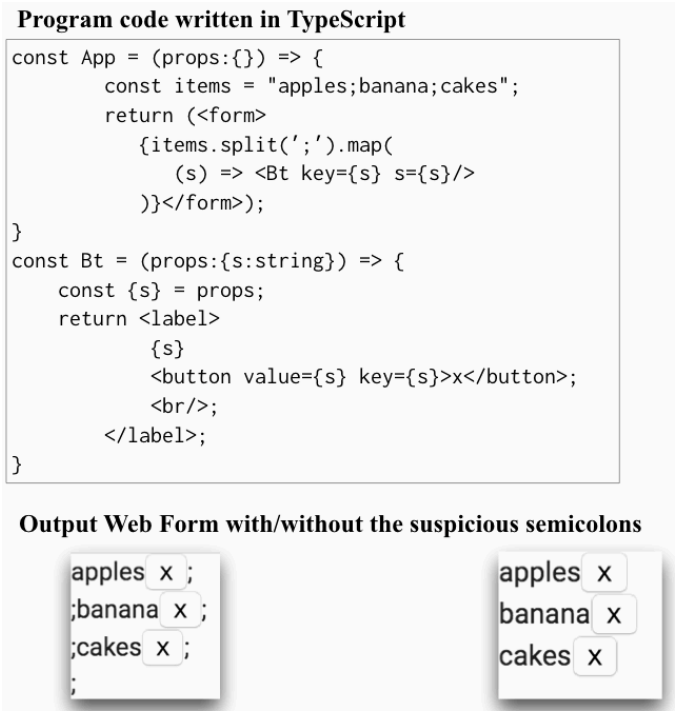

Fig. 5. A React program with a bug

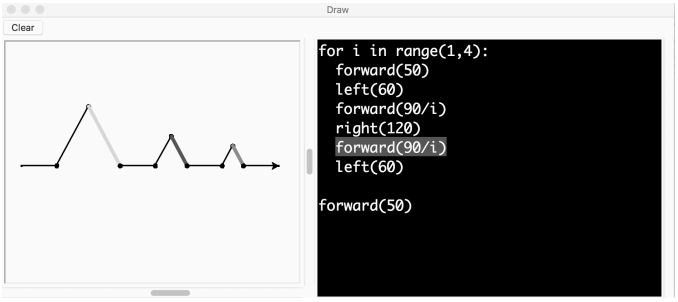

Fig. 6. The turtle graphics tool

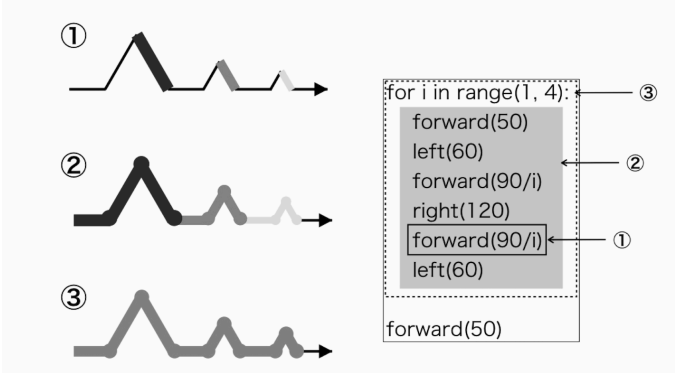

Fig. 7. Expansion and shrinkage of selected area in the turtle graphics tool

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
1  global t, m
2  m ← [ ]        (m has the information for the correlation.)
3  def run ():
4      p ← program text in the editor
5      t ← abstract syntax tree of p
6      id ← 0
7      forall node n of forward statement in t :
8          d ← distance argument of n
9          nn ← a new node of the statement myfd(d, id)
10         id ← id + 1
11         replace n with nn in t
12     exec(t)
13 def myfd (d, i):
14     forward(d)
15     s ← the last segment in the viewer.
16     nn ← the node of myfd(d, i) in t
17     m[s] ← nn
```

Fig. 8. Pseudo code of the correlation in the turtle graphics tool

B. Implementation of the Correlation

This section gives brief sketch of implementation of the correlation in our tool. Our implementation supports any statements including conditional statements and loop statements in Python and Python turtle library. We note that user defined functions and user defined classes are not supported yet.

Fig.8 illustrates the process of the correlation. While Fig.8 is a pseudo code, it is actually written as Python program. Our tool collects the code/output correspondence in runtime. This information is stored at the global variable m. The function `run` in Fig.8 indicates the execution function; first, it obtains the abstract syntax tree of program text in the editor, then replaces every node of `forward` statement with the function call of `myfd` with additional second argument of the node ID. The function `run` finally executes the modified tree t by `exec`, the builtin function in Python. The key point of this pseudo code is that global variables and user defined function are visible during the execution by `exec` function. When Python virtual machine finds a function call of `myfd`, it executes the user defined function at the lines 13–17, where forward statement is executed, and obtains the segment s (fragment of figure) in the viewer. At the line 16, `myfd` function finds the tree node nn of itself; the node ID is used as the search key. Finally, it makes the correspondence information of s and nn.

V. Discussions and Future Works

The technique proposed in this paper correlates statements in a given program code with parts of output. Previous sections illustrate that the code/output correspondence by this technique is useful in program visualization. Since the information simply consists of links between statements and parts, the correspondence can be represented by the highlight effect on program and output viewers. That is, for the users, our technique does not increase the amount of information on screen, while some of the previous works on program visualization show detail and a lot of information on runtime state that has a risk to make novice programmers confused.

On the other hand, output is merely a part of runtime state. The information obtained by our technique may not be sufficient to reason program behavior in some type of programs. One of the future works is studying what type of programs is suitable as a target of our technique. In addition, Fig.8 shows the correlation process, whose target is not covered any user-defined functions. To support user-defined functions in the implementation of our technique is also a future work.

The previous section introduced the tool for turtle graphics. Turtle graphics is often mentioned with fractals drawn by recursive function calls. We need to study visualization of recursive function call and fractal figures with the correlation. Evaluation of the turtle graphics tool is also required.

In section II, we assume the output is a collection of parts. There are several kinds of output like character stream, byte stream and GUI components. Another future work is to extend our technique to support various type of output.

VI. Conclusion

This paper described a technique to make the code/output correspondence in a give program code, and showed how the usefulness of the technique for program understanding. We also reported a development of a turtle graphics editor as an application of our technique.

References

[1] M. Terada, "Etv: a program trace player for students." in *ITiCSE*, vol. 37, 01 2005, pp. 118–122.

[2] P. J. Guo, J. White, and R. Zanelatto, "Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 79–87.

[3] "Python tutor — visualize python, java, c, c++, javascript, typescript, and ruby code execution," (Date last accessed 20-Dec-2018). [Online]. Available: http://pythontutor.com/

[4] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 2481–2490.

[5] P.-Y. P. Chi, S.-P. Hu, and Y. Li, "Doppio: Tracking ui flows and code changes for app development," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 455:1–455:13.

[6] R. Satoh, F. Shituki, and J. Tanaka, "Supports for understanding gui programs using ide with a visualization system of execution," in *Proceedings of 15th Workshop on Interactive Systems and Software*, 2007, japanese edition.

[7] J. Hoffswell, A. Satyanarayan, and J. Heer, "Augmenting code with in situ visualizations to aid program understanding," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 532:1–532:12.

[8] "24.1. turtle — turtle graphics," (Date last accessed 20-Dec-2018). [Online]. Available: https://docs.python.org/3.6/library/turtle.html

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)