Making the Invisible Visible in Computational Notebooks

Mauricio Verano Merino
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
m.verano.merino@vu.nl

L. Thomas van Binsbergen
University of Amsterdam
Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Mazyar Seraj
Eindhoven University of Technology
Eindhoven, The Netherlands
m.seraj@tue.nl

2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) | 978-1-6654-4214-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/VL/HCC53370.2022.9833148

***Abstract*—Notebooks are increasingly popular programming tools adopted by a diverse range of users, including professional and novice users, from various fields not necessarily skilled in software engineering, to experiment with programming and develop software. Notebooks are often used within interactive and exploratory programming settings; however, some of their main use cases are not naturally supported by their design. For example, users can only get insights into the program's state by executing program fragments and updating one's mental model. This paper discusses the possibility of defining widgets to improve notebooks by providing direct insights into the program state. The widgets are developed upon previous work in which a novel approach to incremental programming is suggested based on the notion of an exploring interpreter. As example, we present widgets for visualizing execution history and variable assignments, thereby reducing the cognitive load on users.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

End-user development (EUD) is a human-computer interaction field in which methods and technologies are studied to enable users to extend or customize their software [1]. EUD has received much attention in recent years due to its focus on empowering non-professional programmers, from various domains to create software. There are different ways to support EUD, such as the development of software languages (e.g., high-level programming languages and domain-specific languages), development tools (e.g., IDEs, REPLs, and computational notebooks), and development frameworks. There is an enormous potential for EUD since end-users significantly outnumber professional programmers [2].

Computational notebooks are cell-based documents that allow end-users to interleave prose, code, and results in a single document [3]. Notebooks have become popular in various disciplines such as mathematics, data science, and education. They lower the entry barrier to programming for novices [4], compared to traditional IDEs or plain-text editors employed by professional programmers. There are more than 60 notebook platforms [5], with the most popular provided by the Jupyter project [6].

Although notebooks offer manifold features for end-users, little attention has been paid to displaying feedback of the notebook's state [7] and they miss some valuable features offered by traditional IDEs, such as debugging. These features might empower end-users further, when available in a user-friendly manner.

This paper presents the benefits of using a so-called 'exploring interpreter' as the back-end for notebooks and explains how to develop interactive *widgets* interacting with an exploring interpreter. An exploring interpreter is a bookkeeping device on top of an existing interpreter that keeps track of program state by maintaining an execution graph of reached states and allows reverting to previous states to explore alternative paths [8], [9]. This paper presents initial results of a study into the extent to which exploring interpreters are sufficient to support exploratory styles of programming in programming environments for end-users. As examples widgets we discuss a visualization of the execution graph and a variable watcher, both helping the end-user visualize the program state to better predict the effects of subsequent code executions.

II. Widgets in Computational Notebooks

We use the word 'widget' to refer to an interactive GUI component capable of displaying the program state and enabling the user to manipulate the program state through user-actions. This section presents two widgets, an *execution graph widget*, and a *variable watcher widget*. These widgets are developed on top of a notebook interface based on an exploring interpreter. Therefore, the widgets have access to all the effects produced by executing a code snippet, represented as the difference between the configurations before and after the execution.

The Calc language is a tiny calculator language, and it is used to illustrate the use and benefits of developing widgets on top of an exploring interpreter.

The code snippet below presents the definition of the configuration for the Calc language; it encapsulates the effects that must be stored after executing a valid program. More precisely, this configuration contains the environment (`env`), the possible output produced by the interpreter (`output`), and the result of evaluating the current code snippet (`val`). The environment `env` stores all available variables in a dictionary that uses the variables' names as keys and the variables' content as values.

```
data Config =
   config(Env env, list[int] output, int val);
```

A. Execution Graph

After executing a code fragment REPLs and notebooks present program output and some also present a summary of the (other) effects of the execution to the programmer, thereby

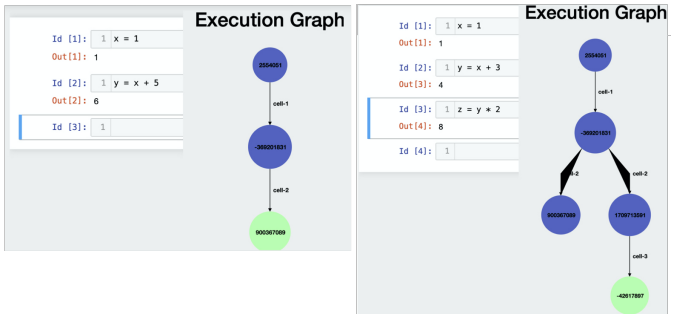

Fig. 1. Resulting execution graph for the Calc language with a single execution path from the root node (left) and the execution graph using an alternative path obtained from selecting a previous node as the current node (right).

enabling the programmer to update their mental model of the program state and making predictions about the effects of the next code fragment [8]. In our notebooks based on exploring interpreters, the effects of code executions manifest as changes to the 'current configuration'. The exploring interpreter maintains version history over configurations by storing configurations in a graph structure. The *execution graph widget* provides a visual representation of this execution graph. In the execution graph (see Figure 1), nodes represent configurations resulting from executing code snippets (programs), and the edges represent program executions. The effect of a program, labelling an edge, can then be defined as the difference between the target and source configuration of the edge. The widget enables users to 'travel in time' by selecting a previously encountered configuration as the execution context for the code cell executed next.

When a new notebook is created, the *execution graph* has a single node (root node), representing the initial (empty) configuration, and it has no edges. When users execute a code snippet, the interface receives the snippet and the current configuration. The notebook interface then calls the exploring interpreter to evaluate the snippet and return an updated configuration with all the effects produced. Currently, the node labels display the configuration's id in the execution graph. To improve the execution graph's readability, its edges are labeled with the corresponding cell number, and the node representing the current configuration is highlighted in green, as shown in Figure 1. Each execution creates an edge in the graph from the current configuration to an updated configuration. If the updated configuration is the same as an existing one, no new node is created, only the edge connecting the nodes, but a node is created if the new configuration does not exist in the graph.

The left-hand side of Figure 1 shows an example of a Calc notebook containing two code cells. The first one assigns the value 1 to a variable `x`. Hence, the execution graph creates a node and an edge from the root node (initial configuration) to the new node (configuration obtained after executing the first cell). Then, the second cell assigns the value of the expression `x+5` to a variable `y`. Since this is the latest execution, the last node is highlighted in green as the current

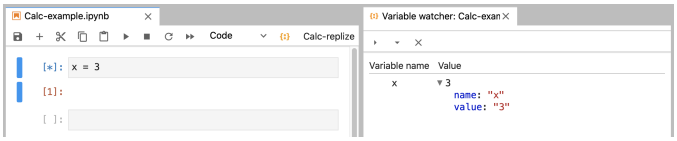

Fig. 2. Variable watcher for the Calc language. Its information is always up-to-date because it is updated when the underlying interpreter's state changes its execution context.

node (configuration). However, the user has found that the value of `y` is incorrect, and instead of `x+5` should be `x+3`. To make this change, the user clicks the second node (the result of executing the first cell) in the execution graph to use that configuration as the current one. After this, the user changes `y` and executes the second cell again. As a result, the current node has edges pointing to two nodes, the old one and the new one, as shown on the right-hand side of Figure 1. If users want to try a different alternative, they can change the current configuration to one of the older configurations, as explained earlier.

B. Variable Watcher

The execution graph widget discussed in the previous section is a powerful tool that, besides exploratory programming, has the potential to reveal the full execution state and history to the user. In this section, we discuss the *variable watcher widget* as an example of a more fine-grained widget providing insights into the current execution state. The variable watcher allows users to read the assignments made to variables, including information about assigned objects (e.g., global variables) and the types or sizes of variables. This widget is useful to end-users to mitigate commons mistakes, such as variable duplication, related to otherwise hidden program states. The current widget (Figure 2) can be extended to become an interactive GUI, following prior work, in which users get the flexibility to create/edit programs using both code and GUI [10], [11]. To support this flexibility, coordination between code/GUI is required, making it necessary to have a common shared underlying state [11]. For instance, the variable watcher can be extended to become interactive and allows users to use CRUD operations. This provides a different interactive interface for end-users, which the underlying exploring interpreter naturally supports, yet this is out of this paper's scope.

III. Conclusion & Future Work

This work presents the initial results from a study into novel features that support exploratory styles of programming in computational notebooks intended in particular for end-user development and domain-specific language users. In this study, the features are supported by so-called exploring interpreters that maintain execution history. This paper has reported on the development of two GUI widgets: a visualization of the execution graph and a variable watcher. Future work is to describe additional general-purpose and domain-specific widgets based on exploring interpreters to be evaluated with the Cognitive Dimensions Framework [12], [13] and user tests.

ACKNOWLEDGEMENTS

The work in this paper has been partially supported by the Kansen Voor West EFRO project (KVW00309) *AMdEX Fieldlab* and the province of Noord-Holland, and has been executed as part of the Agile Language Engineering collaboration[1].

REFERENCES

[1] H. Lieberman, F. Paternò, M. Klann, and V. Wulf, *End-User Development: An Emerging Paradigm*, pp. 1–8. Dordrecht: Springer Netherlands, 2006.

[2] D. J. Rough and A. Quigley, "End-user development of experience sampling smartphone apps -recommendations and requirements," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 4, June 2020.

[3] M. Verano Merino, J. J. Vinju, and T. van der Storm, "Bacatá: Notebooks for dsls, almost for free," *Art Sci. Eng. Program.*, vol. 4, no. 3, p. 11, 2020.

[4] A. Rule, *Design and Use of Computational Notebooks*. PhD thesis, University of California San Diego, 2018.

[5] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo, "The design space of computational notebooks: An analysis of 60 systems in academia and industry," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–11, 2020.

[6] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team, "Jupyter notebooks - a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (F. Loizides and B. Scmidt, eds.), (Netherlands), pp. 87–90, IOS Press, 2016.

[7] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? pain points, needs, and design opportunities," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2020.

[8] L. T. van Binsbergen, M. Verano Merino, P. Jeanjean, T. van der Storm, B. Combemale, and O. Barais, *A Principled Approach to REPL Interpreters*, p. 84–100. New York, NY, USA: Association for Computing Machinery, 2020.

[9] D. Frölich and L. T. van Binsbergen, "A generic back-end for exploratory programming," in *The 22nd International Symposium on Trends in Functional Programming (TFP 2021)*, vol. 12834 of *LNCS*, Springer, 2021.

[10] B. Hempel, J. Lubin, and R. Chugh, "Sketch-n-sketch: Output-directed programming for svg," in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, (New York, NY, USA), p. 281–292, Association for Computing Machinery, 2019.

[11] M. B. Kery, D. Ren, F. Hohman, D. Moritz, K. Wongsuphasawat, and K. Patel, "mage: Fluid moves between code and graphical work in computational notebooks," *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, Oct 2020.

[12] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, 1996.

[13] T. Green and A. Blackwell, "Cognitive dimensions of information artefacts: a tutorial," in *BCS HCI Conference*, vol. 98, pp. 1–75, 1998.

[1] http://gemoc.org/ale/