Introducing a Dataflow visual programming language for understanding program execution

Marin Aglić Čuvić

Faculty of Science, University of Split

Abstract

Understanding program execution is mandatory if a programmer is to write code, regardless of programming experience. Therefore, it is important for novice programmers to construct correct mental models of the execution of the notional machine. To this end, many program visualizations have been developed over the years. However, novice programmers often focus on learning the syntax of a programming language rather than programming itself. Dataflow visual programming languages (DFVPL) allow us to build programs by connecting blocks with arcs. In this paper we present our own DFVPL that exhibits a high level of responsiveness to user inputs and allows the user to control the execution of the program.

Introduction

In order to learn programming, novices often need to learn the syntax and semantics of a programming language and develop an understanding of how programs are executed at the level of abstraction provided by the language itself [1], [2]. This makes learning to program overwhelming for students, and most often they focus on learning the syntax rather than learning to program. Furthermore, novices often develop partial or incomplete understandings of programming concepts which impede the acquisition of new knowledge and may cause students to fail in solving their programming tasks. Researchers have been investigating programming misconceptions for decades, all the while developing new programming languages and tools to facilitate learning to program.

Scratch is probably the most well-known programming language developed for this purpose. Since it is a block-based programming language, where students drag and drop blocks into their appropriate slots, the syntax of the language has been almost eliminated [3]. This allows students to focus on building programs rather than learning the syntax. However, university students quickly outgrow or get bored of Scratch, and require an industry level programming language.

Other than programming languages, researchers have developed program visualizations which graphically depict how programs are executed. The purpose of these systems is to provide novice programmers with the correct mental model of the *notional machine*. The notional machine is an abstract "construct formed from concepts provided by

the programming language" [4] which provides a sufficiently detailed understanding of program execution [5]. A mental model of a notional machine is a mental representation of that machine held by a programmer.

As the before mentioned programming language Scratch, Dataflow Visual Programming Languages (DFVPL) are block based languages connected with arcs (also called wires) where data flows between blocks [6], [7]. A program in a DFVPL has a graph like structure. Like Scratch, DFVPLs almost completely eliminate the need for learning the syntax of a language. Additionally, they make it simple for the user to see the transformation of data as it flows, similar to how program visualizations depict the state of variables. Since DFVPLs make it easy to construct programs, there have been numerous applications of DFVPLs like construction of user interfaces, image processing, music, graphics, general-purpose programming and education [7].

Although these languages make it simple to construct programs they don't allow the user to inspect how these complex functions are actually implemented. Furthermore, only a small subset of them, e.g. Show and Tell [7], are appropriate for novice programmers to learn programming. We believe that some of the key features of program visualizations included into a DFVLP may facilitate learning to program. Therefore, we present a DFVLP that i) allows the user to control the execution of the language while visualizing which expressions are currently executed, ii) explicitly displays the control-flow of a program, and iii) can blocks are conceptually similar to Python instructions.

Program visualizations and education

When writing programs, programmers consult their mental model of the notional machine which allows them to understand and make inferences about the behaviour of the program [5]. However, with novice programmers these mental representations are often faulty and incomplete which prevents them from solving their programming tasks. They are often formed intuitively based on analogies and previous encounters with similar systems. The problem is that, while it is simple to construct a mental model, changing it takes much more effort [8]. Therefore, it is necessary for teachers to provide a correct mental model before learning to program. Program visualizations have been developed for this purpose.

In general, program visualization refers to the use of graphical elements to depict the execution of the notional machine [1]. Since the notional machine is formed from the concepts of a programming language, each language may have its own notional machine. In his literature review [9], Sorva identified forty-six different program visualizations from 1979

to 2013, with J. Hidalgo-Céspedes [1] expanding the list with those that appeared from 2013 to 2016. Some of the most well-known program visualizations include UUhistle [10], Online Python Tutor [11] and Jeliot 3 [12].

UUhistle is a visualizations system that visualizes program execution, but also allows users to take the role of the machine and simulate program execution [10]. Online Python Tutor is a web-based visualization system that currently supports eight programming languages (if we count Python 2 and Python 3 as different languages) and may be embedded into other web pages [11]. Online Python Tutor further has a live programming mode which updates the graphical elements of the visualization as the user types code which allows users to see the changes in program behavior in real-time. Jeliot 3 is a visualization systems developed for many years now whose goal is to facilitate the learning of both procedural and object-oriented programming [12].

All program visualizations allow the users to write and visualize their own code, to step through program execution by step, and show the current state of variables during program execution. These are the core features that allow a system to visualize program execution and allow the user basic interactivity with the system which is important as novices may want to return to a point in a program and repeat some steps. Therefore, we incorporated these features in our DFVPL.

It is worth to note that it is necessary to keep the visualizations simple and avoid having too much animations. As Moreno found in the case of Jeliot 3 - too much repetitions may reduce an animation to a "movie of moving boxes"[13] where students no longer think about what is happening with the notional machine and may miss out on their meaning.

Dataflow visual programming languages

Dataflow visual programming languages (DFVPL) have been researched for more than three decades [14]. DFVPLs are block-based languages where blocks (also called nodes) are connected with arcs (or wires). Therefore, a program in a DFVPL is a directed graph through which data flows between blocks and each block provides a function that may transform the received data [6], [7]. However, it is possible to have blocks that accept no inputs data and those that produce no output data.

Different DFVPLs have been developed for various application domains with blocks providing functions specific to the intended use of the DFVPL. In a way, this is similar to the concept of a notional machine. For example, Orange3 [15] is a DFVPL that provides block for statistical analysis and machine learning. Hence, blocks provide high level functions such as,

e.g. training neural networks or displaying data in a dataset. This allows a simpler way of constructing programs that, in the case of Orange3, analyse data. Other application domains of DFVPLs have been mentioned in the introduction of this paper.

In his paper, Hils [7] grouped DFVPLs according to their application domain and a number of design alternatives. In regards to DFVPL design alternatives, we give a brief summary on only two: i) modes of execution and ii) level of liveliness, as these are the most important for this discussion.

Modes of execution

Execution of nodes in a DFVPL may be data-driven or demand-driven [7]. In data-driven execution, nodes execute as soon as data on the input nodes becomes available. In this execution mode, the data flows downstream. In accordance with the terminology from [7], downstream nodes are those that are found by following a node's output arcs, and upstream nodes are those that are found by going backwards via node's input arcs. In contrast, in demand-driven execution, a node that needs to execute requests data from another node's output arc. If needed, that node requests data from upstream nodes through its input arcs and waits until the data becomes available. Once the data becomes available, the node sends it through its output arcs to other nodes.

In a data-driven execution mode, all nodes are executed, even though some of the computations weren't used [7]. In contrast, in a demand-driven execution mode, only the nodes whose data is requested are executed. However, demand-driven execution is more complex and used less often.

Levels of liveliness

Liveliness is measured on a four-level scale [7]. At the first, "informative" level, visual representations are used as documentation for the program. The second level is termed "informative and significant". At this level, visual representations of the program is executable. In regards to the second level, the third one is enriched with responsiveness, which means that the program is executed each time the user enters input data or edits the program. Finally, at the fourth level are systems that are "live". A system at this level of liveliness continually updates its display to show new data that is being processed as well as results.

A Dataflow Visual Programming Language for novice programmers

At the Faculty of Science in Split, we have developed a DFVPL prototype aimed at helping novice programmers understand program execution. The DFVPL is built with JavaScript and completely web-based.

When designing our DFVPL we wanted to i) have a system that can be used to demonstrate Python programs ii) allow the user to control the execution of program in a step-by-step fashion iii) have a system with a high level of liveliness.

We will discuss these design decisions in the next subsections.

The case for Python

Python is a general-purpose programming language that is often used at universities in introductory programming courses due to its simple syntax. Since the language is popular among introductory programming courses, we wanted our DFVPL to be conceptually similar to Python. One of the problems that we faced was the implementation of simple branching statements into our language. Branching as it is done in Python doesn't correspond well to the dataflow model used by our language, there is no data flowing in an if-else statement (not including tertiary operators). Furthermore, dataflow languages typical use blocks such as merge and switch for branching (due to space limitations, see reference for details). Therefore, our solution was to implement special control-flow blocks that correspond to if, if-else, elif and elif-else statements. These control-blocks do not allow data-flow, and hence must be connected to variables or other nodes that may produce data, or whose data can be inferred while parsing.

Furthermore, the way certain blocks function has been adjusted to simulate Python statements, e.g. print. When the print block is used, it will print out the result in a special program output area. Print can also have an arbitrary number of inputs which are then printed in accordance with their y-position in the workspace.

The Graph Engine

Our next goal when designing our DFVPL was to allow users to control the execution of the program in a step-by-step fashion with a visual enhancement that will display which blocks are currently being executed. This is something that is typical in program visualizations where currently executed lines are pointed to or highlighted. This isn't typical in DFVPLs. Furthermore, when dealing with control structures such as if-then-else statements,

we wanted to display to the users that a certain branch won't be executed. Program visualizations handle this by taking into account which branch is going to be executed when calculating the total number of steps for the user.

This lead to the development of a component we call the Graph Engine (GE). The Graph Engine is the central component of the DFVPL that is made out of a parser, execution service and background execution service.

The input to the parser is the program the users construct. The parser then transforms the graph into a key-value pair data structure. Each key is a step number, and the value contains all of the nodes that will be executed at that step. Therefore, the number of keys determines the number of steps. In case a for loop is found during parsing, the parser includes the loop body the appropriate number of times into the resulting Map.

During parsing, when a branching node is discovered, the background execution service is called. The task of the background execution service is to mark arcs as inactive or active, depending on which ones will be executed by the user. Nodes that are downstream from inactive arcs are ignored by the parser.

Finally, the execution service is the component of the GE that executes the nodes when the user steps through the program. If the steps forward through the program, the execution service executes only the next step (the previous being already executed). In case of going backwards through the program, default values are restored to the nodes coming after the new step.

The liveliness of the system

As discussed earlier in this paper, there are four levels of liveliness. Our DFVPL re-parses and re-executes the program each time the user enters a new value or modifies the program. The execution service re-executes the program up to the current step of execution at which the user is at. Not only that, but all of the visuals and variable values are updated accordingly. Given the discussion presented earlier, our DFVPL falls at the third level of liveliness at least.

Conclusion

In this paper we gave a brief discussion about notional machines, program visualizations. We discussed the importance of constructing a correct mental model of the notional machine. Afterwards, we discussed Dataflow Visual Programming Languages (DFVPLs) and some of their features. Finally, we presented the DFVPL that is being

developed at the Faculty of Science in Split. The main features are that: i) block are functionally very similar to Python statements, ii) the system is at a high level of liveliness, and iii) the system supports some of the features typical for program visualizations.

References

1. J. Hidalgo-Céspedes, G. Marín-Raventós, and V. Lara-Villagrán, "Learning principles in program visualizations: a systematic literature review," in Frontiers in Education Conference (FIE), 2016, 2016.
2. A. Gomes and A. J. N. Mendes, "Learning to program-difficulties and solutions," in International Conference on Engineering Education, 2007, pp. 1–5.
3. J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," Trans. Comput. Educ., vol. 10, no. 4, p. 16:1--16:15, Nov. 2010.
4. M. A. Cuvic, J. Maras, and S. Mladenovic, "Extending the object-oriented notional machine notation with inheritance, polymorphism, and GUI events," in 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2017 - Proceedings, 2017.
5. J. Sorva, "Notional machines and introductory programming education," ACM Trans. Comput. Educ., vol. 13, no. 2, pp. 1–31, 2013.
6. W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in Dataflow Programming Languages," ACM Comput. Surv., vol. 36, no. 1, pp. 1–34, Mar. 2004.
7. D. D. Hils, "Visual languages and computing survey: Data flow visual programming languages," J. Vis. Lang. Comput., vol. 3, no. 1, pp. 69–101, Mar. 1992.
8. R. M. Schumacher and M. P. Czerwinski, "Mental Models and the Acquisition of Expert Knowledge," in The Psychology of Expertise, New York, NY: Springer, 1992.
9. J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," ACM Trans. Comput. Educ., vol. 13, no. 4, 2013.
10. J. Sorva and T. Sirkia, "UUhistle: a software tool for visual program simulation," in Proceedings of the 10th Koli Calling International Conference on Computing Education Research Koli Calling 10, 2010, pp. 49–54.
11. P. J. Guo, "Online python tutor: Embeddable web-based program visualization for cs education," in Proceedings of the 44th ACM Technical Symposium on Computer Science Education, 2013, pp. 579–584.
12. A. Moreno, N. Myller, and E. Sutinen, "Visualizing Programs with Jeliot 3," in Proceedings of the working conference on Advanced visual interfaces, 2004, pp. 373–376.

13. A. Moreno and M. S. Joy, “Jeliot 3 in a Demanding Educational Setting,” Electron. Notes Theor. Comput. Sci., vol. 178, pp. 51–59, 2007.
14. S. Gauvin, M. Paquet, and V. Freiman, “Vizwik – visual data flow programming and its educational implications,” in Proceedings of EdMedia: World Conference on Educational Media and Technology 2015, 2015, pp. 1594–1600.
15. “Orange3.” [Online]. Available: https://orange.biolab.si. [Accessed: 30-Apr-2018].