Testudinata: A Tangible Interface for Exploring Functional Programming

Kritphong Mongkhonvanit
Claire Jia Yi Zau
Chris Proctor
Paulo Blikstein
Stanford University
Stanford, CA 94305 USA
zau@stanford.edu
kritphon@stanford.edu
cproctor@stanford.edu
paulob@stanford.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

Copyright held by the owner/author(s).
IDC '18, June 19–22, 2018, Trondheim, Norway
ACM 978-1-4503-5152-2/18/06.
https://doi.org/10.1145/3202185.3210762

CCS Concepts
•**Human-centered computing** → *Interaction devices;* Collaborative interaction; •**Social and professional topics** → *Computational thinking; Informal education;* •**Applied computing** → *Interactive learning environments;* Collaborative learning;

Author Keywords
Functional Programming; Computer Science Education; Tangible User Interface

Abstract
Learning to program is difficult for most children. Most of the interfaces designed to help children experience and understand programming are based on imperative programming. However, early exposure to functional programming have been found to have many benefits over imperative programming. We describe a tangible interface, Testudinata, that helps to make a fundamental concept of functional programming – function composition – more approachable to younger learners in elementary and middle school. Using Testudinata, learners can design, implement, and test various compositions of pre-made functions on a tangible user interface (TUI), while observing and comparing results on a graphical user interface (GUI). Through the combination of a TUI and GUI, the learners will be able to gain basic understanding

of of function composition in a fun and engaging way.

Background

Functional Programming

A programming language creates a framework to translate ideas into machine instructions, often by providing a model of computations as a mean for thinking about how to solve a problem. Imperative programming casts the computer in the role of sequential instruction follower and emphasizes control of sequential operations. In the case of functional programming, programs are constructed through composition of functions[4]. Functional programming encourages thinking about the problem itself – the what – rather than the sequential nature of the underlying computing engine – the how[5].

Unfortunately, learning functional programming can be a daunting tasks for beginners, especially children. There are many programming environments designed for children, but most are designed for imperative programming. Those with have functional capabilities often do not actively encourage the use of functional programming. For example, SNAP! is a visual programming language similar to Scratch that has several functional programming facilities such as first-class functions and full closures. However, programming in a functional style in SNAP! requires the leaner to be careful not to accidentally mutate the state of the program, a task that is likely to be difficult for beginners.

Why Learning Functional Programming Matters

Functional programming advocates claim that the paradigm leads to programs that are more concise, written quicker, and more robust. This makes them more amenable to formal reasoning and analysis and can be executed more easily on parallel architectures[3]. These properties have positive implications for learning. Fisler found that students who are taught functional programming before imperative programming demonstrate better performance on Soloways Rainfall Problem. This is because functional programming encourage the use of higher-level structures and avoidance of IO, which significantly reduces the possibility of making many kinds of common errors[1].

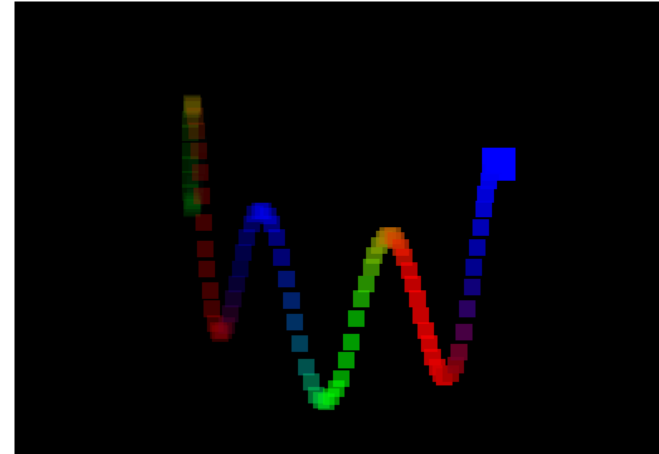

Figure 1: An artwork made in Testudinata

Tangible User Interface

Horn et al. found that, for informal programming activities, tangible user interfaces (TUIs) are more inviting than graphical user interfaces (GUIs) for women and children, meaning that they are more likely to choose to interact with it. They also found that, in general, people are more likely to collaborate actively with TUI, and children are more likely to take a leading role in the activity[2].

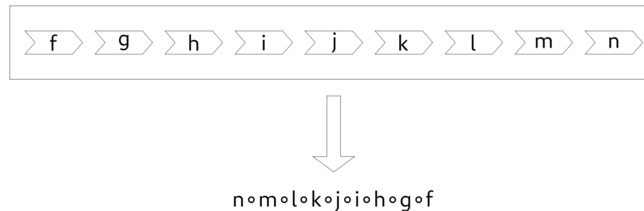

Figure 2: Diagram showing relationship between the position of function plates and the resulting composition of functions

Design

Programs in Testudinata are built purely by composing pre-made functions together. Each function accepts position and time as parameters, modify them in some way, and returns the resulting position and time. These functions can be divided into two main categories: motion function and time functions. Motion functions describe movements that allow the user to display a shape on the screen. When two movement functions are composed together, it produces a result equivalent to adding the parametric equations describing their movements together. All movement functions are made to repeat every one second to make it easier to predict the result of compositions.

Time functions modify the time for functions that follow it. Speed up and slow down functions increase and decrease the rate of the flow of time for all functions that follow it. The delay function adds an offset to the time. This causes subsequent movement functions to produce coordinates slightly earlier than they otherwise would have.

Testudinata's tangible user interface is comprised of a pipe and a set of plates. The plates represent functions that the learner can compose together to build programs.

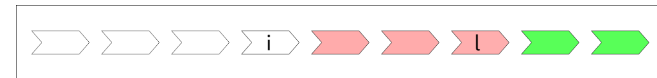

Figure 3: Diagram showing how plates are affected by time functions. Suppose function i multiplies the time by 3, and function l adds 1, then at time $t = 2$, the white plates will get $t = 2$ as one of their inputs, the red plates will get $t = i(2) = 2 \cdot 3 = 6$, and the green plates will get $t = l(i(2)) = (2 \cdot 3) + 1 = 7$.

The learner builds the program by putting the plates in slots on the pipe. This interface makes it easy to re-arrange functions in different orders, encouraging the learner to explore different compositions.

Testudinata is intended to be used as a "cultural building material" similar to LOGO. Although it is more restricted than LOGO in general because programs need to be built by composing pre-made functions, it is more expressive within its domain; it is possible to create complex shapes by composing just a few functions, making it a more powerful as an artistic tool. That said, Testudinata still remain usable for some non-art tasks. For example, in the clock example given by Papert[6], learners could use Testudinata to create clocks by creating motions that repeat with a certain interval.

Future Work

One way to improve on the current design would be to add functionality for examining intermediate values of functions, which will encourage the learner to observe how the result of the composition of functions evolves as it passes through each function. We initially planned to make it possible to insert probes into slots between the function plates, and have the screen display the result of the functions up to the point where the probe is inserted.

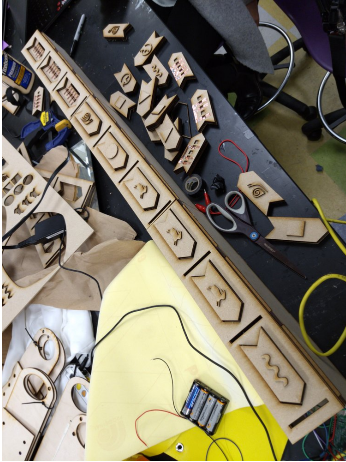

Figure 4: Testudinata's tangible user interface

We had to leave this part out of the design due to time constraint and implementation complexity.

Conclusion

Testudinata is a simplified programming environment with a tangible interface that aims to give learners exposure and basic intuition for functional programming in an engaging and approachable way. It allows learners to construct programs by composing pre-made functions together to generate various kinds of shapes. Through the process of experimenting with various combinations and orderings of functions as well as getting constant immediate feedbacks from the changed in the displayed shapes, the learners gains familiarly and basic intuition for function composition, a fundamental concept in functional programming.

Acknowledgements

We would like to thank Richard Davis and the TAs for Beyond Bits and Atoms: Designing Technological Tools course at Stanford University for their guidance and support.

REFERENCES

1. Kathi Fisler. 2014. The recurring rainfall problem. *Proceedings of the tenth annual conference on International computing education research - ICER 14* (2014). DOI: `http://dx.doi.org/10.1145/2632320.2632346`
2. Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J.k. Jacob. 2009. Comparing the use of tangible and graphical programming languages for informal science education. *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09* (2009). DOI: `http://dx.doi.org/10.1145/1518701.1518851`
3. Paul Hudak. 1999. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, New York, NY, USA.
4. John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. DOI: `http://dx.doi.org/10.1093/comjnl/32.2.98`
5. Milena Vujosevic Janicic and Duan Toi. 2008. The role of programming paradigms in the first programming courses. (2008).
6. Seymour Papert. 1987. Computer Criticism vs. Technocentric Thinking. *Educational Researcher* 16, 1 (1987), 22. DOI: `http://dx.doi.org/10.2307/1174251`