2023 RISP International Workshop on Nonlinear Circuits, Communications and Signal Processing (RISP NCSP 2023), Feb. 28 - Mar. 3, 2023

Dynamic Visualization of Basic High-Order Functions for Learning Functional Programming

Hidekazu Shiozawa[1], Takumi Shimura[1], Koki Asakawa[1] and Takafumi Tanaka[1]

[1] Tamagawa University
6-1-1 Tamagawa-gakuen, Machida, Tokyo 194-8610, Japan
E-mail: {shiozawa,tanaka_t}@eng.tamagawa.ac.jp

Abstract

Functional programming is a paradigm of computer programming that composes a program with functions without side effects like mathematical functions. While it has the advantage of affinity for describing parallel processing, large-scale numerical computation, signal processing, and so on [1], it is considered difficult for even programmers to learn functional programming because of its high level of abstraction. In recent years, however, many programming languages have begun to incorporate basic functional features. We are developing a visual learning system for novice programmers of functional programming. This paper proposes a new method for dynamically visualizing the behavior of the most useful basic higher-order functions including lazy evaluation, and reports on an implemented prototype visualization system. In this method, a function is represented as a horizontal line segment instead of a node or a box, and function evaluation is represented as moving arguments down over the line segment instead of static line connections between nodes.

1. Introduction

The concept of functional programming began with early research in LISP, and dedicated programming languages such as ML, Haskell, and Scala have been put to practical use. It describes computer operations in terms of functions in the mathematical sense without side effects, which always return the same values when called with the same arguments. Functional programming is also characterized by the fact that changing a value of any variable is prohibited in principle. It has been used in academic research and certain business areas because of its simplicity and suitability for parallel processing.

While functional programming has great advantages such as simplicity of description, it is considered difficult to learn due to its high level of abstraction and has not been widely adopted in actual software development. Its concept is very different from procedural and object-oriented models, where the state of a virtual computer is changed by instructions.

In recent years, with the development of compiler technology conventional programming languages such as Java and Python now offer functional syntax and features as a useful complement to conventional programming methods. As a result, functional programming techniques are being introduced even in environments where only conventional programming languages are used. Today, knowledge of functional programming has become a necessity for developers.

However, functional programming is considered to be difficult to learn because it has many unique features.

- Recursive functions: Instead of loops recursions are used for repetitive processing.
- First-class functions: Functions can be treated like data and assignable to variables.
- Higher-order functions: Functions can be used as arguments or return values of functions.
- Currying: Converting a function that takes multiple arguments into a higher-order function of one variable that returns a function.
- Lazy evaluation: The evaluation of an expression is performed at the point in time when the value after the evaluation is needed.
- Closure: A type of function that preserves the context (state) in which it was generated.
- Monad: A monad is used for input/output and performs time-series processing or exceptional processing.
- Map, Reduce, Filter: These higher-order functions are often used for batch processing of data sequences.

Against this background, there is a need for education and teaching materials for basic knowledge of functional programming in information technology universities, technical schools, and software companies.

In programming education, the effectiveness of visual learning materials [2][3] such as dynamic visualization of the program execution process (e.g., algorithm animation) and visual programming that combines visual components such as blocks has long been recognized.

Therefore, with the goal of developing a visual learning environment for functional programming, we are developing a method for visualizing the dynamic behavior of programs that are considered particularly difficult for beginning programmers to understand, such as higher-order functions and lazy evaluation. A higher-order function is a function that treats another function as its argument or return value, and lazy evaluation is an evaluation method in which an expression or function is actually evaluated only when a value is needed in the program. we are considering using visualization methods that are familiar and effective for beginners in conventional programming materials.

2. Related Work

Many program visualizations and visual languages of functional programming have been proposed [1][4][5], but most of them represent the relationship between functions as a network structure with nodes and links. In such methods, it is difficult to represent the dynamic behavior of functional programming, such as higher-order functions and lazy evaluation.

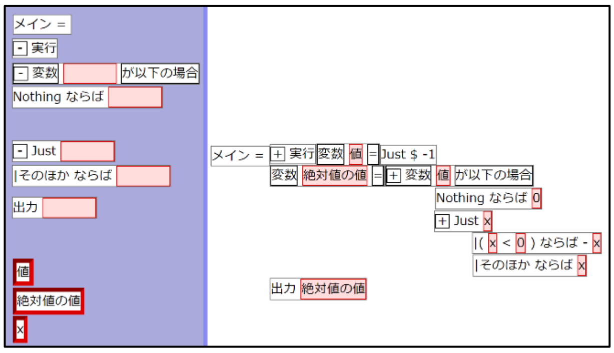
Learning programming often involves writing programs and learning processing concepts from the execution results. Okamoto et al.[6] conducted an experiment in which they visualized the processing of a program written in C using four methods: visualization, identification, predictability, and separation. As a result, it was confirmed that there was a learning effect for more than one-third of the users.

Oshiro et al. [7] developed an elementary algorithm visualization system and proposed an experiment in which they presented a program in a programming language with which the learners were unfamiliar in terms of execution and process structure, and then asked the learners to write the corresponding source code.

In addition, Takeuchi et al. [8] proposed a learning support method that automatically generates fill-in-the-blank questions for functional programming languages and judges whether the answers are correct or incorrect. The results suggest that writing programs in other languages for the same process and comparing them can improve the learning effect.

3. Proposal

To support the learning of functional programming languages, we propose a learning system that dynamically visualizes the processing flow using animation by graphically representing functions and data lists as arguments used in functional programming languages. Among them, we develop a method to dynamically visualize higher-order functions and lazy evaluation



```
メイン = + 実行 変数 値 = Just $ -1
         変数 絶対値の値 = + 変数 値 が以下の場合
                               Nothing ならば 0
                               + Just x
                                   |( x < 0 ) ならば - x
                                   |そのほか ならば x
         出力 絶対値の値
```

Fig1: Functional programming learning environment

mechanisms commonly used in functional programming. The target programming language is Haskell, and examples of Haskell source code are presented.

As a related project, we are developing a visual programming learning environment for beginners in functional programming [9], as shown in Figure 1. It supports a Scratch-like Japanese visual programming language that is interchangeable with Haskell code.

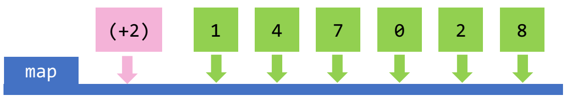
In conjunction with this learning environment, we propose a method for dynamically visualizing the behavior of the most useful basic higher-order functions on data lists in functional programming, such as “map”, “filter”, and “fold” (or “reduce”) and are implementing a prototype visualization system. These higher-order functions take arguments of a function and an array of data such as numbers and apply the function to each element of the array.

Figure 2 and Figure 3 show examples of our proposed visualization method for high-order functions, “map” and “filter” on a data list. In this visualization method, a function is represented as a horizontal line segment instead of a node or a box, and function evaluation is represented as the movement of list elements down over the line segment instead of static line connections between nodes. In addition, by using animation, it is possible to represent the dynamic behavior of higher-order functions, including lazy evaluation. Figure 4 shows an example of a sequence of functions and lazy evaluation. Only evaluated list elements are moved.

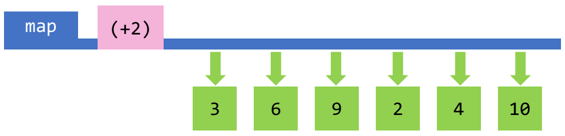
4. Prototype System

Figure 5 shows an example screenshot of the prototype system. When a user enters parameters of a small program, such as function names, variable names, arguments, the system generates a diagram according to the input data and dynamically visualizes the processing flow of four basic functions: “map”, “filter”, “foldl” and “take”, with animation. At the same time, the system generates Haskell source code according to the input.

In Figure 5, there is a data list based on the values entered

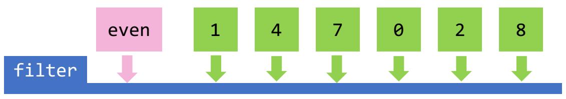


(a) Before function application

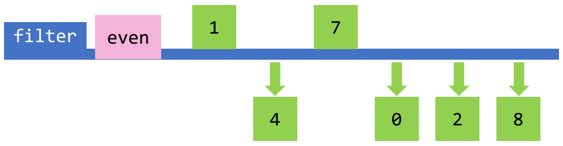


(a) After function application

Fig2: Proposed dynamic visualization of the map function



(a) Before function application



(b) After function application

Fig3: Proposed dynamic visualization of the filter function

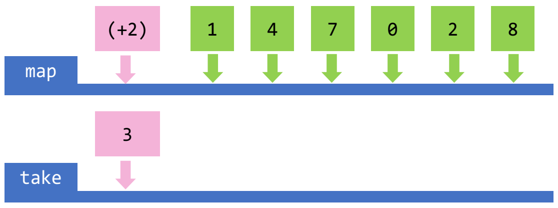
into the system by the user, two horizontal line segments representing the functions, and Haskell code generated by the system. A data list that appears above a horizontal line segment indicates that the function receives a list as an argument that has not been evaluated.

The system expresses that the function evaluates the elements of the argument list by moving each element down over the corresponding horizontal line segment, as shown in Figure 6. In this figure, the filter function outputs a list of elements greater than 2 that pass through the horizontal line segment, and the output list is given as an argument to the print function, which displays the elements.

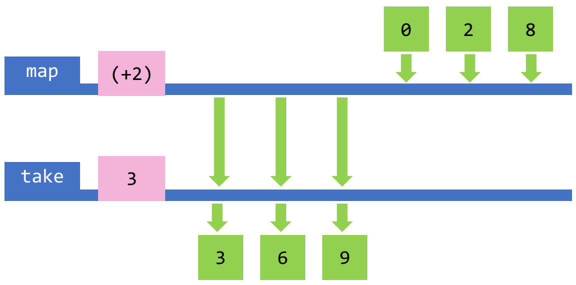
In this system, only those elements whose values are actually evaluated in the lazy evaluation fall in the animation to represent the timing of the evaluation. As shown in Figure 6, elements that do not fall as the animation progresses indicate that they have not yet been evaluated.

5. Summary

The paper reports on our ongoing development of a system that dynamically visualizes basic functions used in the functional programming language Haskell. This system



(a) Before function application



(b) After function application

Fig4: Proposed dynamic visualization of lazy evaluation

provides dynamic visualization of the basic process flow of higher-order functions and lazy evaluation.

For future work, it is necessary to implement functions that could not be implemented in this study and to express properties of functional programming languages such as referential transparency. A user study is also needed.

In this research, we proposed and implemented a method for dynamically visualizing the behavior of basic higher-order functions in functional programming. This visualization will be a part of learning environment for college students in computer science or related departments and software engineers in information technology companies.

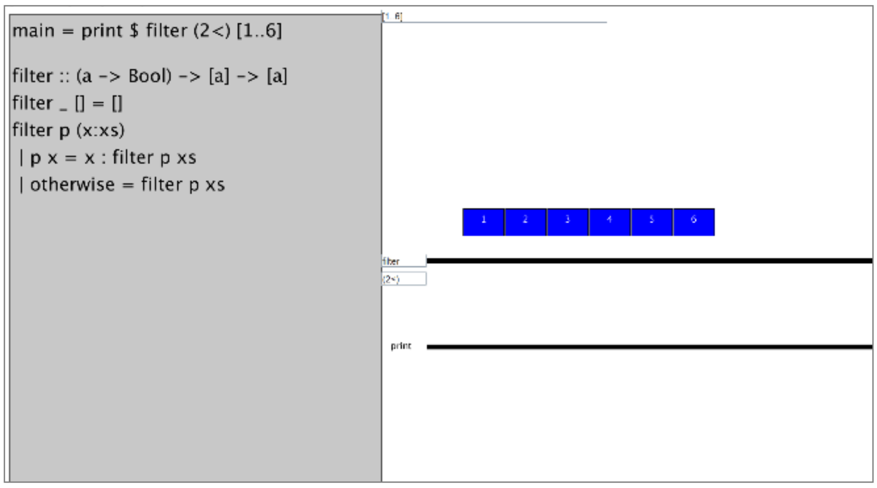
Acknowledgment

This work was partially supported by JSPS KAKENHI Grant Number JP22K12320.

References

[1] H. J. Reekie, Realtime Signal Processing: Dataflow, Visual, and Functional Programming, Ph.D thesis, University of Technology at Sydney, 1995. https://ptolemy.berkeley.edu/~johnr/papers/thesis.html

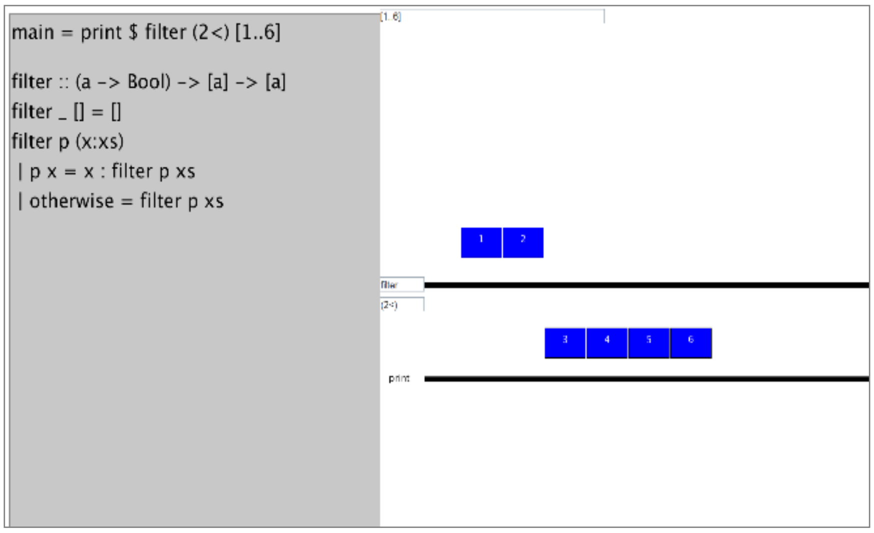
[2] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," ACM Trans. on Computing Education, vol. 13, no. 4, 2013.



```
main = print $ filter (2<) [1..6]

filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
 | p x = x : filter p xs
 | otherwise = filter p xs
```

Fig5: Overview of the prototype visualization system (before the function application)



```
main = print $ filter (2<) [1..6]

filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
 | p x = x : filter p xs
 | otherwise = filter p xs
```

Fig6: Example of the visualization after application and animation of the filter function

[3] T. Tanaka, I. Fujie, Y. Kakara, and H. Hashiura, “A Visualization System of Program Behavior by Animations for Novice Learners,” Proc. NCSP’21, pp.21-24, 2021.

[4] T. Weck and M. Tichy, “Visualizing Data-Flows in Functional Programs,” Proc. IEEE International Conference on Software Analysis, Evolution and Reengineering, pp. 293-303, 2016.

[5] Enso International Inc., Enso (formerly Luna), https://enso.org (accessed Jan. 2023)

[6] M. Okamoto, M. Murakami, N. Yoshikawa, and H. Kita, “Development and Assessment of Learning Materials for Computer Programming on the “Visual Manifestation”,” Japan Journal of Educational Technology, vol. 37, no.1, pp35-45, 2013. (In Japanese)

[7] M. Oshiro and Y. Nagai, “Elementary Algorithm Visualization System for Programming Learning for Beginners,” Proc. IPSJ Information Education Symposium 2018, pp104-111, 2018. (In Japanese)

[8] R. Takeuchi, H. Okubo, and H. Kasuya, and S. Yamamoto, “An Learning Support Environment for Haskell Programming by Automatic Generation of Cloze Question,” IPSJ Technical Report on Software Engineering, vol. 2011-SE-171, no. 15, pp.1-8, 2011. (In Japanese)

[9] K. Asakawa and T. Tanaka, “Visual Programming Environment for Learning Functional Programming Using Unit Test,” Proc. 12th International Congress on Advanced Applied Informatics (IIAI-AAI), 2022.