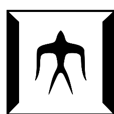


TOKYO INSTITUTE OF TECHNOLOGY

MASTER THESIS

Supporting Recursive Function in Live Data Structure Programming

ライブデータ構造プログラミングにおける再帰関数の支援

Author:
Akio OKA

Student Number:
17M30111

Supervisor:
Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements for the degree of Master of Sciences

in the

School of Computing
Mathematical and Computing Science

February 28, 2019

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Mathematical and Computing Science

Master of Sciences

Supporting Recursive Function in Live Data Structure Programming

by Akio OKA

Live programming is a way to make programming easier. Traditional programming is divided into a phase for editing a program and a phase for confirming whether the program is working as expected. The programmer needs to return to editing the program if the execution result of the edited program differs from the expected result.

Kanon is a live programming environment that automatically visualizes data structures to assist programmers data structure programming. It provides immediate connection between the program text and graphical images of data structures in the programmer's mind. This dissertation reports two studies on Kanon; The one is a user experiment, the other is development of a feature that supports recursive functions.

First, since it was not obvious how the visualization of data structure affects programming, the author carried out a user experiment, which lets participants use one each of the existing Kanon and a text-based live programming environment to solve programming tasks with different degrees of difficulty. Observation of their behaviors and the interview revealed that, though most of the participants had positive impressions, Kanon still has room for improvement.

Second, one of the findings from the experiment is that, though its visualization is useful for the programmer to think about the next code fragment to be written, it becomes useless when defining recursive functions. The author guesses the reason is that, when he or she writes a code fragment following the call to an incomplete function, the object graph displayed in Kanon differs from the state of objects in the programmer's mind. The author proposes an extended feature of Kanon for defining recursive functions.

More specifically, the feature lets the programmer manually build a structure that he or she expects after executing the call. The expected structure serves two roles. (1) It is used as the program state after a function call. When the programmer adds lines after the call, those lines will manipulate the expected structure. (2) It also serves as a test case. Whenever the programmer edits the recursive function definition, the runtime system compares the *actual* data structures from execution against the *expected* structure, and notify the programmer whether they match each other.

This dissertation also discusses the usefulness of the extended feature through case studies.

Acknowledgements

First and foremost, I would like to express my gratitude and thanks to my advisor, Prof. Hidehiko Masuhara, for giving me insightful comments and appropriate feedback. Without his guidance and intuitive ideas, this thesis would not have been completed.

I would like to extend my gratitude to Assistant Prof. Tomoyuki Aotani for sharing his knowledge with me and for providing suggestions on my research. I also grateful to Dr. Jun Kato from National Institute of Advanced Industrial Science and Technology for giving me valuable advice.

Also, I would like to thank all the members of the Programming Research Group for their support and assistance.

Finally, I would also like to express my gratitude to my family for their financial support and warm encouragement.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1
- 1.1 Programming . . . 1
- 1.2 Live Programming . . . 1
- 1.3 Contributions . . . 2

2 Background 3
- 2.1 Data Structure Programming . . . 3
- 2.2 Live Programming Environment . . . 3
- 2.3 Algorithm Animation . . . 4
- 2.4 Kanon: A Live Data Structure Programming Environment . . . 4
  - 2.4.1 Design Overview and Assumptions . . . 4
  - 2.4.2 Visualization of Changes and Two View Modes . . . 5
  - 2.4.3 Backward Connection to Code from Graphical View . . . 6
  - 2.4.4 Control Flow Graph . . . 8

3 Initial Evaluation 9
- 3.1 Design of the Experiment . . . 9
- 3.2 Experimental Procedure . . . 9
  - 3.2.1 Tutorial . . . 10
  - 3.2.2 Practice Phase . . . 10
  - 3.2.3 Main Phase . . . 10
  - 3.2.4 Interview . . . 10
- 3.3 Results . . . 11
- 3.4 Opinions about Kanon's Features . . . 11
- 3.5 Overall Impression . . . 13
- 3.6 Discussion . . . 13
  - 3.6.1 Kanon vs. TLPE . . . 13
  - 3.6.2 Students vs. Researchers . . . 14
  - 3.6.3 Difficulty of the Tasks . . . 14
  - 3.6.4 Is Kanon Helpful? . . . 15
- 3.7 Related Work . . . 15

4 Recursive Function Definition in Live Data Structure Programming 17
- 4.1 Problem . . . 17
  - 4.1.1 Defining Recursive Functions in Traditional Environments . . . 17
  - 4.1.2 Defining Recursive Functions with Existing Kanon . . . 17
  - 4.1.3 Why Is the Visualization by the Existing Kanon Useless? . . . 18
- 4.2 Automated Testing and Overriding for Data Structures . . . 19
  - 4.2.1 Specifying an Expected Structure . . . 19

4.2.2 Automated Testing . . . 20
4.2.3 Object Overriding . . . 20
4.3 Implementation . . . 21
4.3.1 Overview . . . 21
4.3.2 Synchronizing Visualization Context with Cursor . . . 23
4.3.3 Mental Map Preservation . . . 23
4.3.4 Automatic Layout Engine . . . 27
4.3.5 Steadiness . . . 27
4.3.6 Automated Testing and Overriding . . . 28
4.4 Case Studies . . . 32
4.4.1 Reversing a Linked List . . . 32
4.4.2 Insertion Sorting of a Linked List . . . 32
4.4.3 Implementing Interpreter for Lambda Calculus of Tree Structure 33
4.4.4 Result . . . 35
4.5 Related Work . . . 35

5 Conclusion **37**
5.1 Future Work . . . 37

Bibliography **39**

Chapter 1

Introduction

1.1 Programming

Programming is a process to create a *program*, which is an enumeration of characters that gives a computer commands. A person who creates a program is called a *programmer*, and a programmer uses an *editor*, which is one of software, to create or modify a program.

The programming is sometimes a burdensome action for programmers. When the program created by the programmer does not behave as expected, the programmer needs to find which part of the program is wrong. Perhaps, some programmers feel a burden even to write a program.

A lot of research has been conducted to make programming easier. One of the researches is *program synthesis*, which automatically synthesizes an executable program to satisfy given specifications (Gulwani, 2010). Also, there is a research about programming environment which helps a programmer to edit a program. An Integrated Development Environment (hereinafter called IDE), which is a kind of programming environment, provides comprehensive facilities (e.g., editor, build tool, and debugger) necessary to do programming (Teitelman and Masinter, 1981). One of the research to improve programming environments such as IDE is *live programming* (Hancock, 2003; Victor, 2012), which is a way to make programming easier.

1.2 Live Programming

Traditional programming is divided into a phase for editing a program and a phase for confirming whether the program is working as expected. The programmer needs to return to editing the program if the execution result of the edited program differs from the expected result.

Live programming assists the programmer by giving an "immediate connection" between a program and its execution result without requiring the programmer to run the program in their mind. Most past demonstrations of live programming target programs whose results are not obvious from their texts, including the programs for drawing pictures (Victor, 2012), for synthesizing music (Aaron and Blackwell, 2013), for animating game characters (McDirmid, 2007), and for teaching algorithms (Khan Academy, 2018). The details of live programming is discussed in Section 2.2.

Data structure programs (detailed in Section 2.1) fall into the same category, and therefore we believe live programming can be helpful in this domain as well. By

data structure programs, we here mean definitions of data structures and their operations at various levels of abstractions, ranging from generic ones like a doubly-linked list to application-specific ones like "data for a hospital medical record system." In object-oriented programming languages, data structure programs are usually defined as class and method definitions.

Previously, the author proposed a live programming environment, called Kanon, specialized for data structure programming (Oka et al., 2017a; Oka, Masuhara, and Aotani, 2017; Oka et al., 2017b; Oka, Masuhara, and Aotani, 2018). Kanon automatically visualizes data structures to assist programmers data structure programming, provides the immediate connection between the program text and graphical images of data structures in the programmer's mind.

1.3 Contributions

This dissertation reports two studies on Kanon: The one is a user experiment for the initial version of Kanon, the other is development of a feature that supports recursive functions. The contributions of this dissertation are as follows:

- Interview in the user experiment revealed that most of the participants had positive impressions of the initial version of Kanon.
- Through observation of the participants' behaviors, the author observed interesting programming behaviors with Kanon and found that the visualization of Kanon becomes useless for programmers when a programmer defines recursive functions.
- This dissertation proposes a set of features that let the programmers manually specify an object structure expected after the call. The extended Kanon automatically tests whether the actual object structure is the same as the expected structure, and automatically overrides the actual objects to satisfy the expected structure.
- The author confirmed, through case studies, that the proposed features can work well and are helpful when defining typical recursive functions.

In the following chapter, the author discusses the background of this research (Chapter 2). Chapter 3 evaluates the initial evaluation Kanon, and then about the proposing features are discussed in Chapter 4, which includes case studies and implementation. Finally, the author concludes the dissertation (Chapter 5).

Chapter 2

Background

2.1 Data Structure Programming

Data structure programming is the act of programming data structures as well as operations that manipulate those structures. Data structures include common ones like lists and problem specific ones, and appear in many programs.

Data structure programming is sometimes difficult and frustrating, as it is often involved with multiple references. When we define an operation on a data structure, the operation needs to take several steps to modify the structure such as by changing references. In such a case, we need to think about the next step by imagining the shape of the structure modified by the steps written so far. The problem can be even harder when there is aliasing of references and cyclic references.

When we are defining a complicated operation that manipulates a data structure, we sometimes write a test case and examine the (partly) modified structure. However, textual printouts of data structures, which would be the most widely taken approach, are often hard to read, especially when the structures become complicated. It is also difficult to recognize changes in a data structure from its textual outputs.

Though the programmers could have a variety of mental images for data structures, we assume that images with boxes and arrows are common enough. Figure 2.1 is an example of such an image for a doubly-linked list.

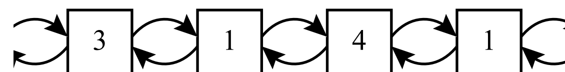


FIGURE 2.1: A mental image of a doubly-linked list.

While it is a straightforward idea and there is a tremendous amount of research that visualizes data structures, it is not obvious what features programming environments should provide in the context of live programming. Though there are many programming environments, like ZStep (Lieberman and Fry, 1995), jGRASP (Hendrix, Cross, and Barowski, 2004) and Python Tutor (Guo, 2013), that visualize user-defined data structures, they mainly focus on the situation when the developer tries to examine the behavior of programs in a *post-mortem* fashion. In other words, development and examination are separated processes in those environments.

2.2 Live Programming Environment

We can classify live programming environments into two groups with respect to the types of the programs they support. The first group's environments enable code editing of a running program (e.g., SonicPi (Aaron and Blackwell, 2013)). The second group's environments automatically re-execute a program to show the effects of

changes immediately (e.g., Live Editor (Resig, 2012) and YinYang (McDirmid, 2013)). The former group is mainly used for artistic performances, like improvising music and animated graphics. The latter is mainly used for software development and pedagogical purposes.

We can also classify live programming by the types of outputs from programs. Many environments mainly target programs that generate *visual or acoustic outputs*. Additionally, environments for artistic performances, demonstrations for educational usages often use programs that draw pictures.

A few exercises are reported to use live programming for programs that do not output visual or acoustic outputs. Live Editor (Resig, 2012) and YinYang (McDirmid, 2013), for example, live-update *textual outputs* from a program being edited. These tools are used to show the course of computation taken place in a loop of a numerical function, such as the square root of numbers.

If we apply live programming to data structures programs, the current environments are not suitable for the following reasons: in the first case, the programmer has to write a program that explicitly generates visual, acoustic or textual outputs. This is clearly tedious for operations that manipulate data structures. In the latter, the only standard way to output data structures is printing in text, which is not friendly to the programmer's eyes as we discussed in the previous section.

2.3 Algorithm Animation

Many algorithm animation systems, including Balsa (Brown and Sedgewick, 1984), Zeus (Brown, 1991) and Tango (Stasko, 1989), can graphically display data structures. Some of these systems provide frameworks where we can easily develop an animation by instrumenting an implementation of an algorithm like sorting.

While these algorithm animation systems could be used for program understanding, they are fundamentally different from live programming, as they do not have a *live updating* feature. In other words, they are designed for visualizing behaviors of completed programs; they would not work well for partly-written and frequently edited programs. Though it would be hypothetically possible to automatically apply an algorithm animation system to a program being edited, it would not provide continuous feedback as we will discuss in the later section.

2.4 Kanon: A Live Data Structure Programming Environment

We proposed Kanon, a live programming environment for data structure programming (Oka et al., 2017a; Oka, Masuhara, and Aotani, 2017; Oka et al., 2017b; Oka, Masuhara, and Aotani, 2018). Here, we introduce notable features of Kanon that are relevant to define recursive functions.

2.4.1 Design Overview and Assumptions

Figure 2.2 is a screenshot of Kanon. The left, upper-right, and lower-right sides are the editor pane in which a program is written, the visualization pane that displays data structures, and the call tree pane that displays control flow, respectively. It is designed under the following assumptions.

- We assume that a program is written in JavaScript[1] in a single file. It consists

[1] We chose JavaScript as a general-purpose programming language that supports data structures. Therefore, we do not consider use-cases specific to JavaScript, such as DOM and async.

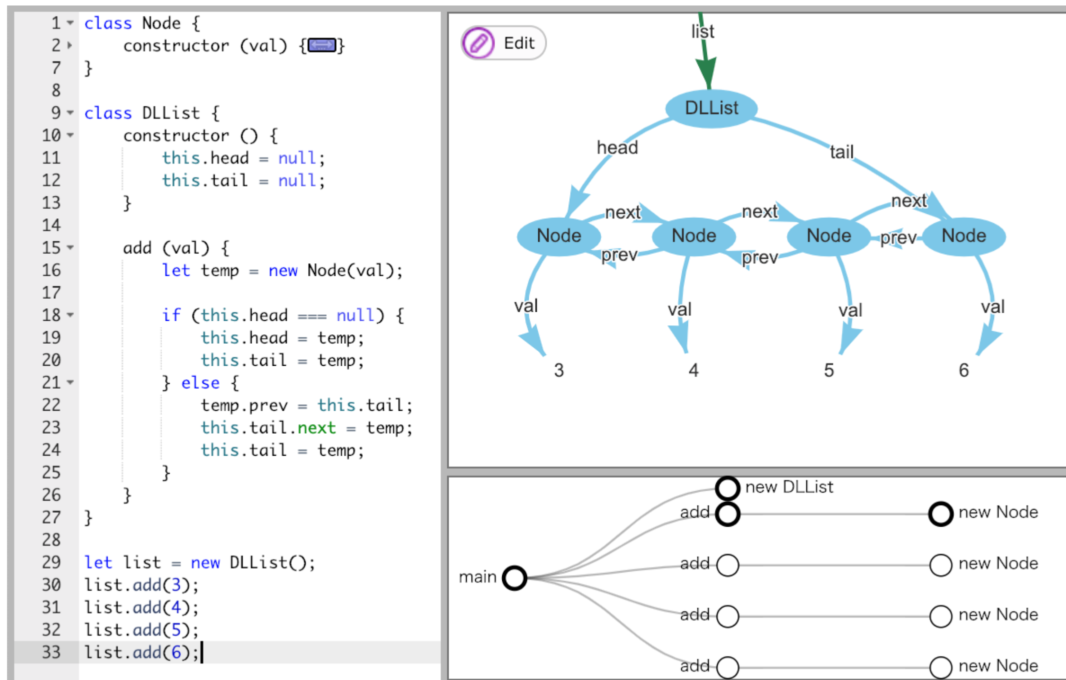


FIGURE 2.2: A screenshot of Kanon.

of definitions of data structures and their operations, followed by top-level expressions that serve as test cases.

- Kanon draws data structures as a node-link diagram. Each oval in the visualization pane represents an object that is created during an execution, labeled with the class name of the object. The blue arrows from the ovals show the field values in the object, which point to either other objects or primitive values, and the green arrows with no origin (e.g. the arrow labeled `list`) show which object the local variables refer to.
- Kanon continuously executes the program, and visualizes all objects created from the beginning up to an execution point that corresponds to the cursor position in the editor.

2.4.2 Visualization of Changes and Two View Modes

2.4.2.1 Visualization of Changes

When a live programming environment visualizes data, the data can change during execution. For example, given a program that repeatedly approximates a mathematical function, we might want to see the changes of intermediate results during a run. Existing environments can show such changes as a series of values (McDirmid, 2007; Imai, Masuhara, and Aotani, 2015) or as a line chart (Apple Computer, 2016). For programs that produce visual images (i.e., drawing programs), there have been attempts to use a stroboscopic visualization (Granger, 2012) or a timeline visualization (Kato, McDirmid, and Cao, 2012).

For data structures, there is no definitive way to visualize changes. Though there have been a number of studies on algorithm animation, those studies tend to develop techniques specialized to specific algorithms.

2.4.2.2 Snapshot and Summarized View Modes

Kanon provides two ways of showing changes in a program run: one is called the *snapshot view mode*, which animates the graphical representation using cursor movement in the text editor, and the other is called the *summarized view mode*, which shows summarized effects of changes in one graphical representation.

With the *snapshot view mode*, the view shows the object graph with variable references when the program execution reaches the cursor position. If the execution reaches the cursor position multiple times (due to multiple function calls or loops), the execution of a specific *context*[2] is chosen.

Figure 2.3 is an example of a snapshot view for the program text in Listing 2.1 (in which the cursor position is denoted by a black rectangle), in the context of `l.add(4)`. The green arrows in Figure 2.3, which are labeled `this` and `temp`, represent the references by the `this` expression and the variables available in the specified calling-context. In this example, the programmer is defining the `add` method for doubly-linked lists and has finished defining the case when the list is empty. The view shows the object graph when the execution of `l.add(4)` reaches the cursor position. Note that the node for 5 is not yet created in this view.

With the *summarized view mode*, the view shows effects of a statement[3] at the cursor position over the object graph at the end of the execution. The view summarizes the effects by two means: (1) when the statement is executed more than once, it visualizes all the effects performed in those executions; and (2) when the statement contains a function call, it visualizes all the effects performed in the call.

Figure 2.4 is an example of a summarized view for the program text in Listing 2.2, where the programmer has finished definition of `add`. This view shows an object graph at the end of execution. At the same time, the view illustrates the effects of the code at the cursor position, in this case, the assignment "`this.head.next = temp;`", where the orange solid arrow (③) shows the reference found at the end of execution. The dashed arrows (①, ②) denote the overwritten references, i.e., once created by this (orange ①) or other (green ②) assignment, and then disappeared due to later assignments.

From the diagram, the programmer can observe that the final graph is incorrect. The `next` field of the leftmost `Node` should reference the middle `Node`, where instead it references the rightmost `Node` in the final state. The programmer can also see that the reference was initially correct (as shown with the green dashed arrow ②) and then overwritten by the assignment at the cursor position. In fact, the cursor line should assign to `this.tail.next`, instead of `this.head.next`.

2.4.3 Backward Connection to Code from Graphical View

2.4.3.1 Relating Visualized and Program Elements

It is important to the programmer to be able to establish a connection between visualized information and program elements. For example, consider an environment that visualizes a time-series of values of multiple variables in a program. We then need to find out the correspondence between a series of values to a variable, as well as a value in a series to a specific moment in an execution.

[2] Users can spcify the context by clicking one node in the call tree.

[3] Though our current implementation only shows effects of one statement, it is not difficult to extend it to show the effects of a series of statements. It is a part of our future work.

LISTING 2.1: Partially defined add.

```
class DLList {...
  // add the given val at the end of the list
  add(val) {
    var temp = new Node(val);
    if (this.head === null) {
      // when the list is empty
      this.head = temp;
      this.tail = temp;
    } else {
      // when the list is not empty
      █
    }
  }
}
var l = new DLList();
l.add(3); l.add(4); l.add(5);
```

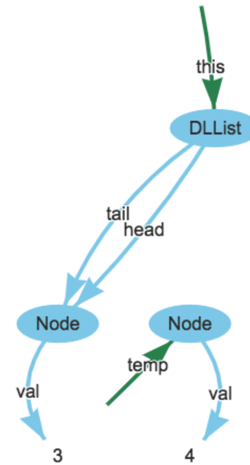


FIGURE 2.3: A snapshot view in the context of `l.add(4)`.

LISTING 2.2: Finished (yet incorrect) definition of add.

```
class DLList {...
  // add the given val at the end of the list
  add(val) {
    var temp = new Node(val);
    if (this.head === null) {
      // when the list is empty
      this.head = temp;
      this.tail = temp;
    } else {
      // when the list is not empty
      temp.prev = this.tail;
      this.head.next = temp;█
      this.tail      = temp;
    }
  }
}
var l = new DLList();
l.add(3); l.add(4); l.add(5);
```

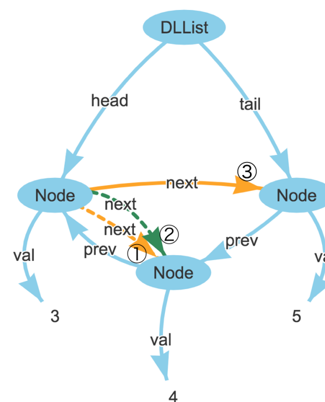


FIGURE 2.4: A summarized view for the program in Listing 2.2.

YinYang's solution to this issue is a *probe* that displays a value of an expression just below the expression, and a *tracing* construct that produces a clickable output, which rewinds the program state to the time when the output is produced.

For data structures, since a visual representation (i.e., a node-link diagram) has a structure, environments should help to establish a connection between those visual elements and program elements.

2.4.3.2 Jump to Construction

We provide a mechanism that helps to connect visual elements to program elements. The mechanism is called *jump-to-construction*, which is invoked by a double-click on a graphical element, and moves the cursor position to the program element that corresponds to the graphical element (either a new expression or a field-assignment statement).

2.4.4 Control Flow Graph

Kanon provides a call tree pane that displays which function calls which function as shown in the lower right in Figure 2.2. The nodes in the call tree represent parts of the program that might be executed multiple times (i.e., function, loop, and constructor).

The user can specify the specific context in the snapshot view mode. As shown in the call tree pane, the border of some nodes is thick. This thick border means that the context is selected for the snapshot view mode. In the case of Figure 2.2, the user has selected the second execution as the `add` method defined in the `DLList` class. Therefore, when the cursor moves into the `add` method in that situation, the visualization pane shows an object graph at the time that the execution reaches the cursor position in the context of the function call at line 32 in Figure 2.2.

Chapter 3

Initial Evaluation

As an initial evaluation, we carried out a user experiment[1] in order to collect programmers' opinions about the implementation of existing Kanon. Since we have not yet implemented many practical features such as code completion, we do not believe that we can do the meaningful quantitative evaluation. (In the experiment, we measured time to task completion, which is not a primary purpose of the experiment.) Nevertheless, as we will see in our experiment, we observed interesting programming behaviors with Kanon, positive opinions on the Kanon's features, and several future improvements.

3.1 Design of the Experiment

In our experiment, we let the participants use Kanon to solve several programming tasks in order to observe their usage of the Kanon's features, and to gather their opinions. In addition, we designed the experiment with the following questions in our mind.

- *Do graphical outputs make difference in programmer's behavior from textual outputs?*
  We build a textual live programming environment (called TLPE hereafter) as a counterpart of Kanon and let the participants use both environments.
- *Does the amount of programming experience affect the usage of Kanon?*
  Our experiment had 13 participants consisting of 9 students (at the senior undergraduate and graduate levels) and 4 computer scientists in a corporate research laboratory. Note that all the students and one scientist have heard about Kanon before the experiment, but none of them have ever used it.
- *Does difficulty of programming tasks affect the usability of Kanon?*
  We prepared two tasks with different difficulties and let the participants solve them. The simple one merely requires to modify a few object references. The difficult one requires to traverse references while modifying the references themselves.

3.2 Experimental Procedure

We carried out the experiment for each participant one by one. The participant took the four phases, namely tutorial, practice, main and interview, which amount to approximately one and a half hours in total. We recorded the participant's activity

[1] This experiment was carried out in Japanese all the time. In this dissertation, the participants' opinions have been translated into English.

by recording the computer's screen and participants' voice. Throughout the experiment, we asked the participants to speak out their thoughts, for example "I'm confused now", "Why is the figure displayed like this?" and "Oh, this program includes an error." What the participants are thinking tells us where they are paying attention to during programming.

3.2.1 Tutorial

In the tutorial phase, the participant is asked to read a 67-pages document that describes the usage of Kanon and the format of the tasks. In this experiment, all the tasks are to define a method for a common data structure. The participants were given the definition of the data structure, a definition of the method without an empty body, and a series of method call expressions that serve as test cases, which cover all the situations. The document uses the scene in which a programmer defines the `LinkedList.add` method as an example. At the same time that the participants read the document, they are allowed to use Kanon to grasp how Kanon works.

3.2.2 Practice Phase

In the practice phase, the participant is asked to define a `LinkedList.insert` method as the exercise simple task using Kanon in order to get used to Kanon and the format of tasks.The excercise task took up to approximately 15 minutes. Throughout this phase, we allowed the participants to question anything. After they have completed the task or time is over, we commented the feedback and the answer to the task to them.

3.2.3 Main Phase

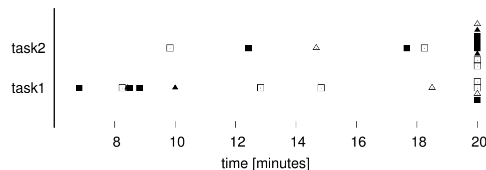
In the main phase, the participant tasked to solve two tasks, namely **rotate** and **reverse**, in this order. We grouped the participants into two, and assigned Kanon or TLPE to those tasks according to Table 3.2. Each task was given a 20 minutes time limit.

The **rotate** and **reverse** tasks are to define a method that rotates a root node of binary tree, and a method that reverses a doubly-linked list, respectively. The former task can be accomplished by merely modifying a few references in the given tree nodes. The latter task is rather difficult, as it requires to follow links between nodes while modifying those links.

TLPE is a simple live programming environment as shown in Figure 3.1. It provides a customized `println` function that prints out data on the right-hand side of the screen. It is live in the sense that the output is immediately updated whenever the code on the left-hand side changes, which is similar to Khan Academy's Live Editor (Resig, 2012) and YinYang (McDirmid, 2013). Unlike the built-in `print` function in JavaScript, the `println` function displays internal elements in a nested data structures. It also supports cyclic structures (by showing #*n* as in the figure).

3.2.4 Interview

In the interview phase, one of the authors asked the participants several questions. The first question is about difficulties throughout the experiment. The role of this question is to clearly remind the participants of their thoughts during the experiment. Then, for each feature of Kanon, we asked the participants their opinion. We



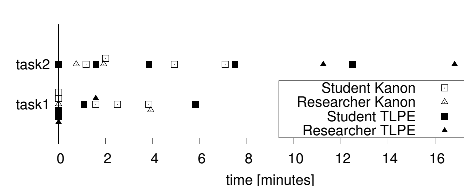


FIGURE 3.1: Textual Live Programming Environment (TLPE).

FIGURE 3.2: Time taken to solve the tasks.

FIGURE 3.3: Error time.

encouraged them to answer, not just "good" or "bad", but rather concrete opinions on specific parts of the feature, and possible improvements. Finally, we asked an overall impression of Kanon. The participants may answer the impression of Kanon itself, or in comparison with TLPE.

3.3 Results

Table 3.1 and Table 3.2 show a quantitative result of this experiment such as time taken to complete the tasks and a count of using features of Kanon. The qualitative opinions received in the interview are described below.

3.4 Opinions about Kanon's Features

About **the snapshot view**, there are several positive opinions like:

- *"The feature was very helpful for changing references"*, and
- *"The feature is reasonable because we often want to see the states (of the program) around the code fragment being written."*

Visualization of variable references (e.g., the arrows labeled `this` and `temp` in Figure 2.3) are also positively taken by most participants, but also had suggestions like:

- *"I also wanted to see arrows for function arguments[2] when I was writing a recursive function,"* and
- *"It looks strange that the arrows for variables lacks originating ovals."*

The summarized view was not used by the most participants. One participant gave the reason:

[2] The current implementation of the *snapshot view* merely displays locally declared variables and `this`, but not function arguments.

TABLE 3.1: Average number of times of uses of each feature. ("Snap:Summ" means the proportion of each in the overall task. The column of "Context" is constructed the number of (correct usage : incorrect usage). Each column of "JtoC" and "print" is the number of using *jump to construction* and print statement, respectively.)

(A) task1

Snap : Summ	Context	print	JtoC
80.6% : 19.4%	2.57 : 3.57	2.17	0.29

(B) task2

Snap : Summ	Context	print	JtoC
96.9% : 3.1%	7.5 : 3.17	6.29	0.17

TABLE 3.2: The participant information.

	Group A	Group B	total	min/ave/max exp	#js
task1 (**rotate**)	Kanon	TLPE			
task2 (**reverse**)	TLPE	Kanon			
#Students	5	4	9	2 / 4.4 / 10	2
#Researchers	2	2	4	14 / 17 / 25	1
#total	7	6	13	2 / 8.3 / 25	3
min/ave/max exp	2 / 7.3 / 14	3 / 9.5 / 25	2 / 8.3 / 25		
#js	3	0	3		

- *"I only needed the snapshot view mode"*.

However, there are opinions that suggest its potential like:

- *"The visualization with orange and green arrows was very easy to recognize,"* and *"I used it to show the final state of the program, and completed the task by imagining the other states,"* (This participant mainly used the summarized view mode.)
- *"It might be useful when it is hard to understand an overview of a program"*, and
- *"It might be useful for tasks of fixing bugs."*

A few participants used the **jump-to-construction** feature but they thought that the feature was not so helpful. The opinions are:

- *"I had no chance to use it,"*
- *"It might be useful for larger programs because it would be difficult to understand the overview,"* and
- *"Construction sites are not so relevant when we modify fields in existing objects."*

The **automatic layout engine** had positive comments like:

- *"The visualization was similar to what I imagined,"* and
- *"The figure after the completion of the task was cleanly arranged without moving the node ourselves."*

At the same time, it also had suggestions like:

- *"Though the final layout looks good, it does not in the middle of programming,"* and
- *"I wanted to undo the (automatic) layout as it became messy."*

3.5 Overall Impression

Overall, the participants gave positive comments like:

- *"It is amazing. I want to use it,"*
- *"With visualization, it was easy to program since I often draw pictures when I reason about data structures,"* and
- *"I felt it is wonderful when I was solving a task with TLPE."*

and also several suggestions for future improvements like:

- *"It was helpful for those tasks, but not sure if it will be so for other situations and for large programs,"*
- *"When the object graph disappears, I wanted to know the reason*[3]*,"* and
- *"After I thought about (the strategy) based on the visualization, I had to think again based on the program. I wish I could generate code fragments by directly manipulating the object graph. "*

3.6 Discussion

This section discusses finding in the results of the experiment as well as the observations of the participants' behavior by the authors.

3.6.1 Kanon vs. TLPE

The experiment did not give a clear answer whether the graphical representation as opposed to the textual representation is useful. With respect to the task completion times on Figure 3.2, TLPE is faster than Kanon for the `rotate` task, or is as fast as Kanon for the `reverse` task. (Again, due to the limitations of the current implementation, we do not consider the task completion times are the primary factor of the experiment. Also, the number of participants is not large enough to evaluate statistical significance of those figures.)

From the closer observations, we had the following findings and insights.

- Roughly the half of the participants misused the Kanon's features related to changing specified context. As the "Context" column on Table 3.1 shows, selection of the execution context seems to be difficult. This suggests that Kanon needs more improvements on its GUI.
- Some participants, when they were using TLPE, drew object graphs on a paper. This suggests usefulness of graphical representation regardless the environment used.
- The styles of problem solving are different by the participants. Some carefully thought out algorithms before writing code, and some others carried out the trial-and-error style programming. Those difference in the styles and the difference programming environment could affect each other, which should be investigated in future.

[3] In the version we used for the experiment, Kanon will erase the object graph when a runtime error occurs. The error message was actually displayed bottom of the screen.

- We observed interesting difference in the participants' behaviors when errors occurred. First, from the observations of the behaviors, we found that many participants took, when using Kanon, longer time to notice occurrence of errors. This would be partly because the current design of the environment that reports errors as a plain text at the bottom of the screen, which hardly attract the programmer's attention. However, this would also due to the policy of our visualization, which keeps showing a previous visual image when an error occurs. This policy is based on the fact that, when the programmer is editing a program, it transiently becomes incorrect either syntactically or semantically. By preserving a previous image, the next image from a successfully executed run is smoothly connected with an animation. At the same time, by seeing the previous image, the participants sometime misunderstood as the program was executed without errors but the object graph was not changed. With TLPE, the programmers can immediately notice runtime errors because an error will prevent execution of subsequent `println` calls, which results in disappearance of the output.

  Second, for the `reverse` task, the participants with TLPE spent longer time with erroneous states. Though we do not have clear explanation for this, one possible reason would be that Kanon helped finding an incorrect state that will cause an error in the subsequent execution.

- Even though we do not observe clear difference in the task completion times, the participants opinions favor Kanon as reported in Section 3.5. We would like to consider the reasons why they thought like that.

3.6.2 Students vs. Researchers

With respect to the task completion times, we did not observe clear differences between inexperienced and experienced participants (i.e., the students and the research laboratory scientists, respectively). This might be because the students took the courses on programming and data structures more recently.

We noticed difference a difference between students and researchers in their programming styles. While the researchers tried to add more test cases on top of the provided cases, the students declare completion by only considering the provided test cases.

3.6.3 Difficulty of the Tasks

Difficulty of tasks and visualization can affect the types of mistakes that the programmer makes. While the types of the mistakes with Kanon and TLPE are not different for the easier task (**rotate**), we observed a unique kind of mistakes with Kanon for the more difficult task (**reverse**).

For the **reverse** task, several participants with Kanon took an inappropriate plan to solve the problem. Since the task is to reverse a doubly-linked list, the function must have a loop scanning over the nodes. A common and correct plan is to let each iteration of the loop modify the forward and backward references of a focused node. However, some participants with Kanon planned to modify the forward reference of the focused node and the backward reference of the next node in one iteration (which was at least not easy to maintain references consistently across iterations, and all of them eventually gave up the plan).

Though we cannot investigate the cause of this mistake, we conjecture that the visual representation might mislead the programmer at the planning of a solution. With Kanon, the programmer starts defining `reverse` with visual representation, where the first and the second nodes reference each other. With this visual clue, one might plan to modify those two references.[4]

In general, we believe that making a correct plan is crucial for solving difficult programming tasks regardless the programming environment used. With a new type of programming environment, we would probably need more experience to develop good recipes for typical types of problems.

3.6.4 Is Kanon Helpful?

In the experiment, we observed rare usage of some features, namely the *summarized view mode* and *jump-to-construction*. We presume that this is due to the type of the tasks used in our experiment, which are to define a new function body. We design the *summarized view mode* and *jump-to-construction* for the situations of modifying a program and of understanding program behavior, respectively. We might have observed more usage with those features if the experiment included such tasks.

As mentioned in Section 3.6.1, notifying the programmer an error as early as possible is crucial. We found that the problem is not trivial. In a live programming environment, a program transiently becomes an erroneous state when the programmer edits a code fragment. The environment should also delay notification so as to smoothly connect visual images between the states without errors. The problem is even more difficult when a program correctly runs around the code being edited, but causes an error at the later execution point.

3.7 Related Work

Collabode is a web-based Java IDE, and it is evaluated by a user study (Goldman, Little, and Miller, 2011). Through the user study, they confirm its usefulness qualitatively and quantitatively. The qualitative evaluation is based on the participants' thought observed in the questionnaire phase.

[4] The current implementation draws nothing for a field with `null`. Hence the first node has only one outgoing reference. This could also be the cause of the mistake.

Chapter 4

Recursive Function Definition in Live Data Structure Programming

In this chapter, we propose features which solve one of the problems found in the experiment. First, we will describe the problems when defining recursive function with existing Kanon. Second, we will propose a mechanism to solve the problems. Third, we will present case studies to show the features are helpful, and then we will explain how to implement them.

4.1 Problem

There is a problem in Kanon when defining a recursive function. Before explaining the problem, we first review a process of defining a non-live recursive function definition.

4.1.1 Defining Recursive Functions in Traditional Environments

Many programs define recursive data structures, whose functions are also recursive. For example, a linked list is a pair of an data element and another linked list. A tree node is a tuple of a data element and children tree nodes.

A recursive function for a recursive data structure usually consists of two parts, namely, the base case and the recursive case. The base case defines operations when the function reaches at an edge (i.e., a tail of a list, or a leaf of a tree) of the structure. The recursive case applies the recursive function to its recursive components, and combines the results.

Listing 4.1 shows an incomplete recursive method `reverse` for linked-lists. The then clause of the `if`-statement defines the recursive case when `this` node has a trailing list. The else clause defines the base case when `this` node is at the end of a list. After receiving a reversed list of the trailing list in the then clause, the programmer will write a code fragment that adds itself to the tail of the reversed list and returns the head of the reversed list, whose code is not shown in the listing.

4.1.2 Defining Recursive Functions with Existing Kanon

Even though it is possible to use Kanon to define recursive functions, it is not as helpful as it is for non-recursive functions. The problem is that the visualized data structures at the cursor position, which supposed to assist the programmers to decide what they should write next, become useless for recursive functions.

Here, we explain the problem by using the same example above. Assume that the programmer created a test case with a linked-list and method call as shown in Listing 4.2, partially wrote `reverse` as shown in Listing 4.1, and then located the

LISTING 4.1: An example of recursive data structures and methods.

```
1  class Node {
2      constructor(val, next) {
3          this.val = val;
4          this.next = next;
5      }
6
7      reverse() {
8          if (this.next) {
9              let next_lst = this.next.reverse();
10             // users assume that the recursive function behaves as
                   expected in order to write here
11         } else {
12             return this;
13         }
14     }
15 }
```

cursor at the end of the recursive function call (i.e., at the end of line 9.) Even though the cursor is located *after* the recursive function call, Kanon shows an object graph (Figure 4.1) that remains the same structure as the one *before* the recursive call.

The visualization is useless, or even harmful, to define the rest of `reverse`. What the programmer should do is to add a code fragment that appends `this` node to the end of `next_lst`. However, in the visualization, the trailing list is not reversed.

LISTING 4.2: An example call expression of `reverse` method.

```
1  let list = new Node(1,
2                  new Node(2,
3                      new Node(3,
4                          new Node(4, null))));
5  list = list.reverse();
```

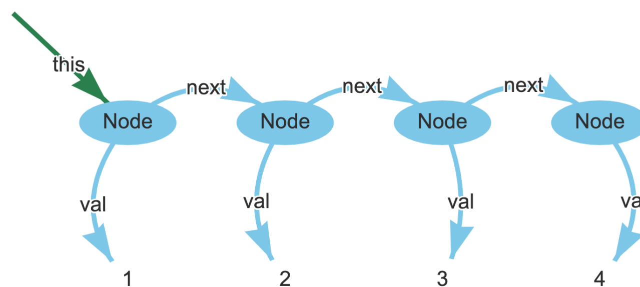


FIGURE 4.1: A node-link diagram displayed by the existing Kanon just after writing the recursive function call in Listing 4.1.

4.1.3 Why Is the Visualization by the Existing Kanon Useless?

The cause of the problem can be generalized as follows. (Note that in the above situation Kanon worked as it was intended.)

The reason is that the object graph displayed in Kanon differs from the state of objects in the programmer's mind. On one hand, the programmer thinks about the

next code fragment by reasoning about (A) the program state that is supposed to be at the cursor position. On the other hand, Kanon shows (B) the program state that is actually obtained at the cursor position. In most cases, Kanon is helpful since both of the states (A) and (B) are the same. However, in the case of a recursive function, they do not match since (A) is a completed program and (B) is an incomplete program the programmer is currently writing.

4.2 Automated Testing and Overriding for Data Structures

We propose a set of extended features for defining recursive functions in Kanon. The features let the programmers, when there is a call to a partly written function, manually specify an *expected* object structure after the call.

The expected structure serves two roles. (1) It is used as the program state after the function call. When the programmer adds lines after the call, those lines will manipulate the expected structure. (2) It also serves as a test case. Whenever the programmer edits the recursive function definition, the runtime system compares the *actual* data structures from execution against the expected structure, and notify the programmer whether they match each other.

4.2.1 Specifying an Expected Structure

We provide a user-interface to specify an expected structure where the programmer directly manipulates a node-link diagram on the screen by using a pointing device. The interface lets the programmer specify in the following two steps.

4.2.1.1 Selection of a Function Call with a Calling Context

First, the programmer selects a function call expression with a calling-context by moving the cursor in the editor pane. When the program executes the selected function call more than once, the calling-context currently used for visualization is chosen.

In the following, we assume that the programmer selects the recursive call at line 9 in Listing 4.1 with the first calling-context; i.e., the first execution of `reverse`, which is called from the top-level call at line 5 in Listing 4.2.

4.2.1.2 Building an Expected Structure

Second, the programmer builds an expected structure. After selecting the function call and calling-context, the system pops up a window showing a node-link diagram. The diagram is the object structure just before calling the function.

Figure 4.2 shows a screenshot of Kanon with a window to edit the graph to specify the expected structure[1]. The window appears after selecting the recursive function at line 9.

The programmer builds the expected object structure by modifying the diagram. When the modification is completed, clicking the *accept* button[2] on the upper right corner of the window registers the diagram as an expected structure[3].

[1]When building an expected structure, the user can move any nodes by dragging.

[2]Clicking the *actual graph* button in Figure 4.2 opens a new window with an object graph just after executing the selected function call. The new window is just to check the actual structure.

[3]Our current implementation does not support a function that merely returns primitive values, though it would not be difficult to support such a function.

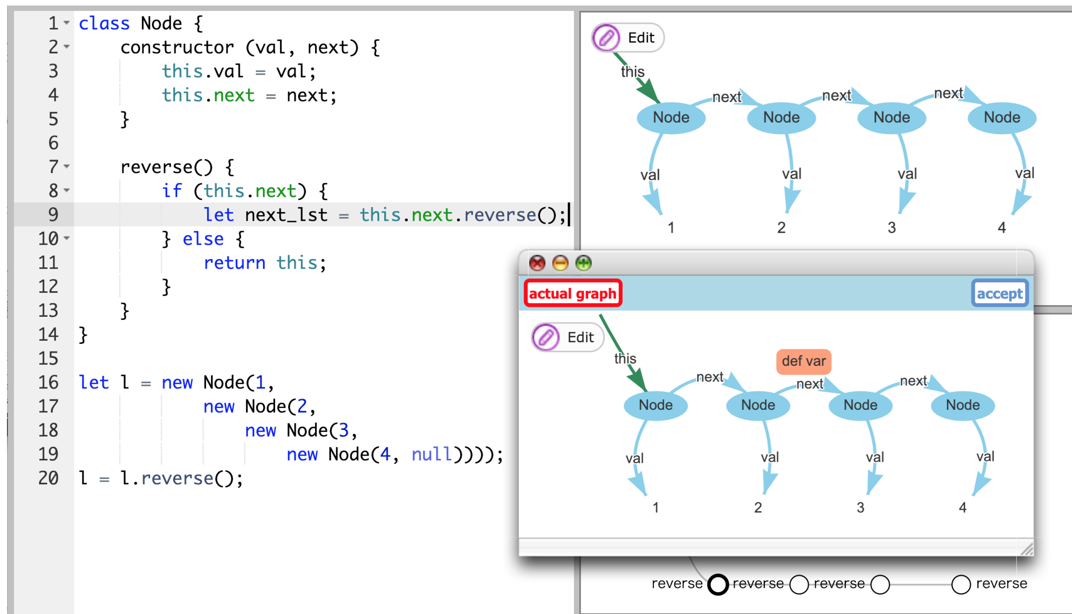


FIGURE 4.2: A pop-up window for building an expected structure. The red rectangle *def var* is a button to introduce a variable reference (like the green arrow with `this`) into the diagram.

4.2.2 Automated Testing

Whenever the user program has no syntax error, Kanon automatically runs the program and checks, at each execution context where a test case is inserted, whether the *actual structure* (i.e., the objects generated by the execution of the program) matches the expected structure. We call it *automated testing*.

When Kanon matches the expected and the actual object structures, it uses *reference equality* for objects existing before the function call, and *structural equality* for objects created during the call. In other words, two references are regarded as equal when they are pointing at the identical object in the object structure before the function call, or pointing at newly created and structurally equal objects.

4.2.2.1 Visualization of Test Results

The result of testing is shown in the editor and the call tree panes. If a test case succeeded, the background color of the parentheses of the respective function call becomes green. Otherwise, it becomes red (as shown in Figure 4.3).

Additionally, the result is also displayed on the visualization of the call tree (like Figure 4.4.) The red circle represents that all test cases at the context failed. The red edge represents that the test of the function call on the calling-context failed. If all test cases in the context succeeded, the circle becomes green. If there are both succeeded and failed test cases in a context, it becomes yellow.

4.2.3 Object Overriding

If Kanon finds a failed test case (i.e., the actual structure differs from the expected structure), it automatically *overrides objects*; i.e., it modifies fields of objects in the actual structure so that they match the expected structure, and let the program continue execution with the modified objects.

```
this.next.reverse();
```

FIGURE 4.3: A function call which an expected graph is inserted and its test failed.

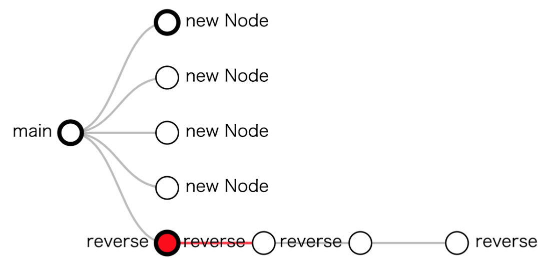


FIGURE 4.4: A call tree with test highlight.

We assume that the programmer built the expected structure as shown in Figure 4.5. From the programmer's viewpoint, Kanon merely shows an object diagram identical to the manually built expected structure as if it were constructed by the execution of the (incomplete) recursive function. However, this lets the programmer to continue *live programming* with the expected structure; i.e., when he or she adds code fragments that manipulate the expected structure after the function call, and can further observe the result of manipulation from the visualization.

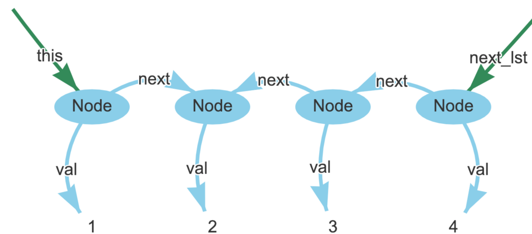


FIGURE 4.5: An example of a graph after overriding object structure.

4.3 Implementation

We implemented a prototype of Kanon for JavaScript running on web browsers. It is available online[4]. Below, we first overview the implementation and then describe how to realize the automated testing and overriding.

4.3.1 Overview

Figure 4.6 overviews the implementation. We will explain the structure by following the operations taken place upon a program modification.

We use a modified version of the Ace editor (Jakobs, 2018) for editing program text. When the programmer edits a piece of text, it notifies the visualization engine.

① The visualization engine uses Esprima (Hidayat, 2018) to parse the program text in the editor. It then traverses the syntax tree by applying the following modifications:

- It inserts declarations of global variables for keeping track of calling contexts and the virtual timestamp.

[4] https://github.com/prg-titech/Kanon (source code), https://prg-titech.github.io/Kanon/ (executable in web browsers)

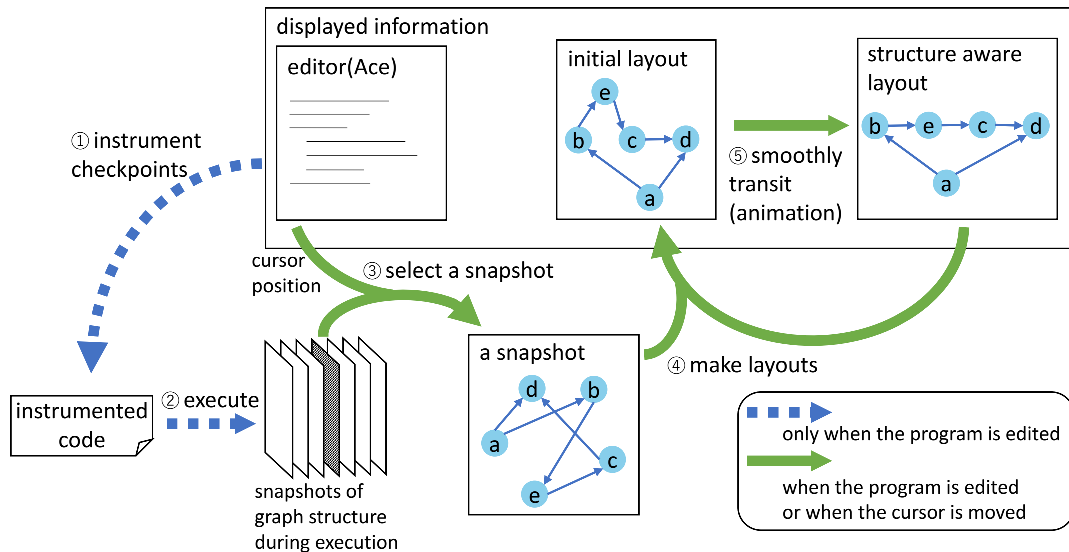


FIGURE 4.6: Overview of the implementation.

- For each `new` expression, it appends a piece of code that records object ID in a special field of the created object.
- For each statement and `new` expression, it inserts checkpointing code before and after the statement. The checkpointing code is an expression that applies a list of global and local variables to the object traversal function.
- At the beginning of each loop body and function body, it inserts counting code.

② The engine then evaluates it using `eval`. When the checkpointing code runs, it collects JavaScript objects that are reachable from the variables in the scope. We use the object reflection mechanism to obtain field values from an object. The objects and their references are recorded as graph data (i.e., nodes and links) with a virtual timestamp that increases every checkpointing execution.

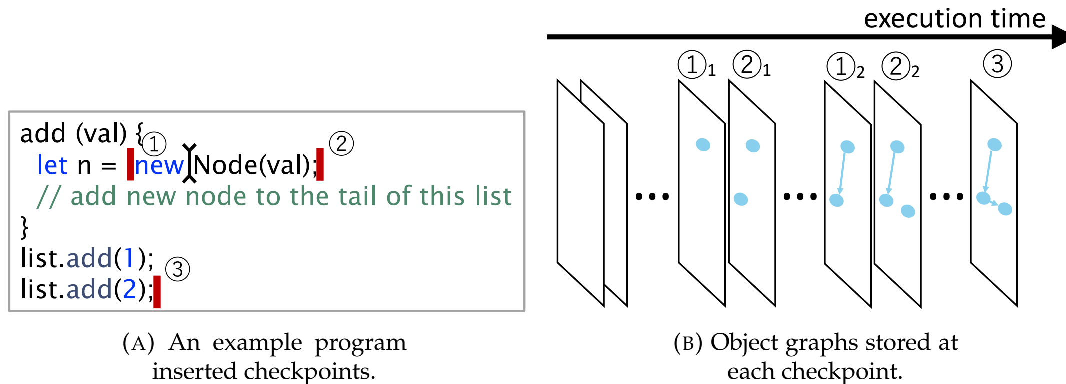
③ To update graphical representation, the engine first obtains the cursor position from the editor and then identifies the nearest checkpoint to the cursor position. It then calculates a range of virtual timestamps which corresponds to the current visualization context. Finally, it selects the object graph that is recorded at the nearest checkpoint within the calculated timestamp range.

④ The engine computes the initial layout of the nodes of the objects in the object graph, and draws the graph. The layout is computed by using the currently shown layout (for an older object graph) in combination with a physics-based graph layout algorithm. First, it determines the set of the nodes in the new object graph that are included in the old object graph. It pins those nodes down to the same geometric locations as in the layout currently shown. Second, it runs a physics-based graph layout algorithm so that the newly created nodes will be placed aesthetically-pleasing positions. We use the vis.js visualization library (B.V., 2018) both for calculating the layout as well as for drawing the resulted layout.

⑤ Finally, the engine computes the structure-aware layout and smoothly moves the visualized graph from the initial layout to the new one. The current algorithm simply recognizes a list or binary-tree structure based on field names, and then places the nodes of the structure on a horizontal line or a tree shape.

4.3.2 Synchronizing Visualization Context with Cursor

Kanon provides two view modes for visualization of data structures which are changed as execution progresses. Here we explain which object structure is selected in these two view modes as a graph displayed in Kanon.



(A) An example program inserted checkpoints.

(B) Object graphs stored at each checkpoint.

FIGURE 4.7: Which checkpoint is chosen?

In the *snapshot view mode*, Kanon displays an object structure stored at the closest checkpoint before the current cursor position. Because the selected checkpoint might be executed multiple times, the user can specify a context of the closest loop or method surrounding the cursor position. In the case of Figure 4.7, "the closest checkpoint before the cursor position" indicates checkpoint ① and Kanon selects ①$_n$ from the specified loop count n.

In the case of the *summarized view mode*, Kanon displays an object structure stored at the final checkpoint. Additionally, Kanon calculates the difference between the object structures stored directly before and after the cursor position for each loop iteration. When the two checkpoints are directly within the same loop or method, we highlight the difference in the graph. In the case of Figure 4.7, we display the object graph stored at the final checkpoint, namely ③, and highlight the nodes and links that are different either between ①$_1$ and ②$_1$, and between ①$_2$ and ②$_2$.

4.3.3 Mental Map Preservation

4.3.3.1 Motivation

Live programming environments should *preserve the mental map* when a program is modified. Here, the mental map[5] means a representation in the developer's mind who saw a visual image of a program output. Preservation of the mental map is achieved, when the system displays a visual image of an output of a new program, by keeping the differences of those visual images as small as possible.

For example, adjusting constant parameters in a drawing program is one of the well-known demonstrations of live programming. By immediately executing (i.e., drawing pictures) a modified program, the programmer can observe the effect of changes as animation. Some environments provide special mechanisms like slider bars for continuously modifying constant values (Khan Academy, 2018).

[5]The concept of the mental map preservation was proposed for drawing algorithms for dynamically changing graphs (Archambault and Purchase, 2012; Lee, Lin, and Yen, 2006). It should not be confused with the concept of *navigability*, which concerns about the connection between a code fragment and a visualization element in live programming environment (Burckhardt et al., 2013). We discuss the features related to navigability in Section 2.4.3.

In Kanon, the mental map preservation means keeping the differences of the graph layouts as small as possible, when it draws a modified object graph. This property is known to be important in the studies of dynamic graph drawing (Archambault and Purchase, 2012; Lee, Lin, and Yen, 2006) because the human who saw a graph would need a lot of time to grasp the structure of the graph.

4.3.3.2 Problem of a Naïve Implementation

It is not a trivial task for Kanon to preserve of the mental map. Assume that a programmer is writing a function `make` that creates a binary tree. Figure 4.8(a) shows an incomplete function definition that merely creates right children of the tree, which effectively creates a linked-list. The programmer moves the cursor at line 9 in order to insert a piece of code, and sets the visualization context to the second call to `make`, where Figure 4.8(b) is the object graph at this moment. Note that *the programmer is thinking about a `Node` object referenced by a `next` field of the root node, whose visual representation is at just right of the root node.*

Now the programmer inserts a statement "`node.left = make(n-1);`" to line 9, which lets the program create a binary tree. The questions are: Where should the nodes of the binary tree be placed? Where should the visualization context be set?

If there were a naïve algorithm that places the nodes created by the modified program based on the order of object creation, its visualization would be like Figure 4.8(c). *The `Node` object (linked with `next` from the root) the programmer was thinking about is now located at a upper-right position from the root* (the dashed oval in the figure). *At the position the programmer was focused on, there is a `Node` object referenced by the `left` field of the root node* because it is created by the second execution of line 8. Another problem is that a context that differs from the context they were focusing on is specified. They must expect the context of function call of line 10 during an execution of function call of line 14 as a context of the snapshot view mode. However, a specified context after the insertion differs from the expected context because an addition of several functions calls by the insertion changes the focused context of function call from second execution to ninth execution.

4.3.3.3 Context-Sensitive Identification for Mental Map Preservation

When a program is modified, Kanon visualizes the new object graph and maintains the context (in the snapshot view mode) so as to preserve the mental map. Here we first explain the requirements for this feature and then describe the proposed mechanism.

Since Kanon executes the modified program under a fresh environment, it needs to identify (1) a visualization context that corresponds to the one previously displayed, and (2) mapping between objects created in the execution of the modified program and those created in the previous program. in Figure 4.8, this means (1) identifying one of seven executions of line 8 that corresponds to the second execution in the previous program, and (2) identifying three of seven `Node` objects that correspond to the ones created in the execution of the previous program.

We propose a novel technique, called *calling-context sensitive identification*, for preserving mental map of object graphs[6]. The technique gives a calling-context based

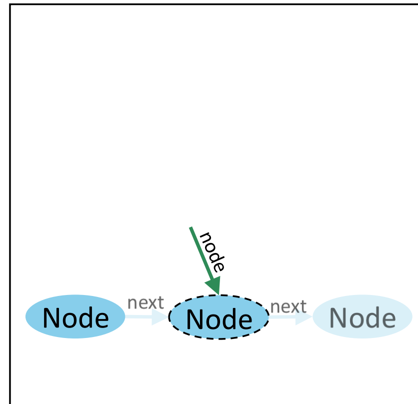
[6]The use of calling-contexts per se is not a novel idea as there are systems that use calling-context for identifying corresponding execution points before and after program modification. The novelty of the paper is the use of calling-context for visualization of object graphs. We discuss the existing systems in Chapter **??**.

```
1  class Node {...}
4  2
5  function make(n) {
6    if (n === 0)
7      return null;
8    let node = new Node();
9    ▌
10   node.next = make(n-1);
11   return node;
12 }
13
14 let tree = make(3);
```

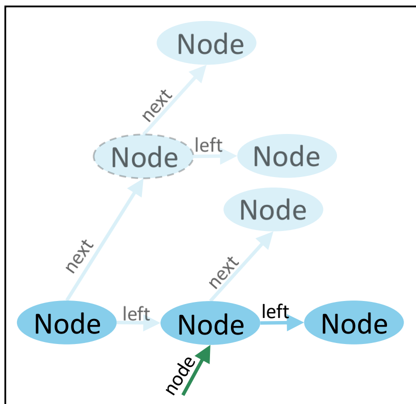
insert

```
node.left = make(n-1);
```

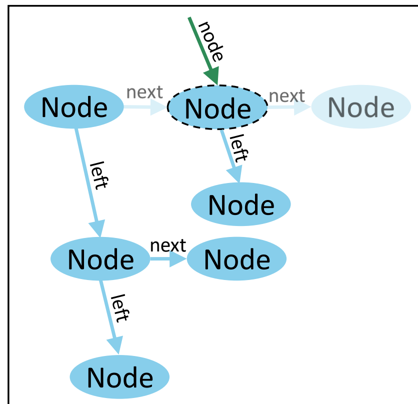
(A) An incomplete program that creates a binary tree.



(B) The visualization before insertion. The dashed node is the programmer's focus.



(C) A naïve visualization after insertion. The position of the dashed node (the `next` of the root) is moved.



(D) A mental-map-preserved visualization. The dashed node stays at the original position.

FIGURE 4.8: Naïve and mental-map-preserved visualizations after code editing. (For the ease of understanding, the nodes that are created beyond the current visualization context are also drawn.)

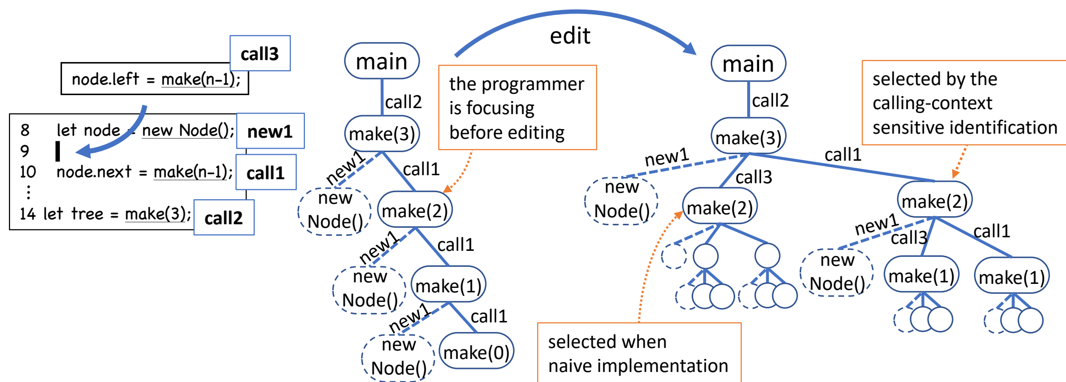


FIGURE 4.9: An example of labeling and simplified call trees before and after editing on Figure 4.8. A `new` expression is also regarded as a node of the call tree.

identifier, called *context-sensitive ID*, to each function call, and records each object graph by associating it with the context-sensitive IDs. When it executes a modified program, it selects an object graph matching the context-sensitive ID, and draws nodes so that the objects with the same context-sensitive ID will be placed at the same positions.

Figure 4.9 shows two call graphs that explain the calling-context sensitive identification. Those call graphs (note that they are *not* Kanon's visualization) respectively represent the executions of the programs before and after the insertion. A node of the call graphs is either a function call or object creation, attributed with a label (e.g., `call1` and `new1`) that denotes a source code location. A context-sensitive ID of a call graph node is a list of the labels on the path from the root node.

Kanon uses the context-sensitive IDs, when a program text is modified, to identify the "same" visualization context and to identify the "same" object. In Figure 4.9, when the current context was the second call to `make` in the older program, the context-sensitive ID of the context is `"call2-call3"`. In the call graph of the modified program, the context that has the same context-sensitive ID is the right child of `make(3)`. This means that the executions triggered by the newly added line are successfully skipped even in the modified program.

In addition, it uses the context-sensitive ID of the `new` expression as the object ID. In Figure 4.9, the secondly created `Node` object in the older program has `"call2-call1-new1"`. When the program is modified, the object that has the same ID in the new call graph will be placed at the same position.

Our proposal can be summarized in this way:

- We give a unique label to each program location (precisely, we only maintain labels for `new` expressions, object literal, method call expressions, and loops.) We preserve the labels by tracking the modification when the programmer edits the program text.

- We give a context-sensitive ID to each execution of an expression such as function call and new expression. In order to determine each object's context-sensitive ID, a stack (each frame of which is configured by either method call label, `new` expression label or a pair of loop label and loop count) manages the context during execution.

- When `new` expression is executed, we use the stack information for context-sensitive ID as the ID of the created object.

- When we draw an object graph obtained from an execution of a modified program, we lay it out so that each object will be placed at the same position of the object with the same ID in the execution of the previous program.
- When a program modification changes the call tree, we change the context for snapshot view mode so that the context after the change is the same as the context-sensitive ID before editing.

Intuitively, we consider two objects to be the same when they are created by the same expression, and the execution of the new expression has the same calling context and the same loop count.

Kanon manages a table of program locations for associating expressions with labels, which is robust for most type of editing operations, but has some limitations. When Kanon finds an expression in a program, it gives a new label and records the beginning and ending locations. Upon an editing operation like insertion or deletion of characters, it shifts those locations if necessary. Therefore, it can identify the "same" expressions before and after editing in most cases. There are some operations, such as cut-and-paste and undoing, that the current implementation cannot keep track of, but we believe some of those operations can also be supported by bookkeeping the labels for the text inside the cut- and undo-buffers in the editor.

4.3.4 Automatic Layout Engine

The current automatic layout engine implemented in Kanon specifies special layout behavior for some structures. Currently, these include binary trees and linked lists. In the case of binary trees, the nodes are specially arranged only if each element is constructed by a `Node` class and the left element and the right element are represented by `left` and `right` properties. In the case of linked lists, the nodes are specially arranged only if each element is constructed by a `Node` class and the next element is represented by the `next` property.

In order to implement the above, it is necessary to identify the specified structures from the set of objects. First, in order to find the root of the tree or the head of the list, we must check both sides of each edge. If the root of the tree or the head of the list is found, we then calculate the position of each element. In the case of binary trees, we position each element so that the distance between elements at the deepest level is kept above a threshold. We then set the horizontal position of the parent element to the center of the horizontal position of its child elements. In the case of linked lists, we position each element so that the distance between elements is kept above a threshold. At this time, we adjust the entire graph to preserve the center of gravity of the elements.

However, this layout engine still needs more improvements. The layout engine should recognize arbitrary data structures other than lists and trees. It should also provide a mechanism to shrink or fold unimportant nodes so that the programmer can see a large data structure within a limited drawing area. Supporting customized visualization, where the programmer can control the presentations of data structures, is also important.

4.3.5 Steadiness

Following the lessons from Hancock and Victor (Hancock, 2003; Victor, 2012), we implemented several mechanisms to stabilize the visualization in Kanon, though

there are still many challenges. We here explain those mechanisms and then present the remaining challenges.

Similar to many live programming environment, it keeps the previous visualization when the program has a syntax error. This prevents the visualization flicks while editing the program text.

When the execution of a program causes a runtime error (e.g., null pointer dereferencing), it either updates the visualization if the error happens after the previous visualization context, or keeps the previous visualization with translucent colors[7]. The former case is useful to see the immediate effect of the code fragment being edited, regardless the future errors. Surprisingly, some live programming environment, such as Khan Academy's Live Editor, does not have this feature. For the latter case, an alternative is not hide the visualization. We did not do so because in Javascript programs often cause runtime errors while they are being edited. For example, when we are typing a long variable name, the program causes an unbound variable error at runtime until we finish typing. Making the visualization translucent in the latter case is important to alert the programmer of the error; otherwise the programmer sometimes misinterpret that the program runs up to the previous visualization context yet no changes were made to the objects.

When it updates the visualization with a new snapshot (either selected by cursor movement or by editing the text), it suppresses re-drawing the object graph as long as the graph is topologically equivalent to the shown one. This is a workaround to avoid the inconvenient feature of underlying visualization library, namely vis.js, that randomly changes the geometric positions of edges every time it draws the same graph.

4.3.6 Automated Testing and Overriding

4.3.6.1 Specifying an Expected Structure

We provide an user-interface to specify an expected structure where the programmer directly manipulates a node-link diagram on the screen. They can apply operations (e.g., adding nodes, changing references of objects) to the node-link diagram displayed on the opened window by using a pointing device.

As a library to open windows in the browser, Kanon uses Prototype Window Class (*Prototype Window Class*). In the window, the manipulation mechanism utilizes the feature provided in vis.js library (B.V., 2018), which enables us directly to manipulate the displayed graph.

If they click the *accept* button in the upper right corner of the window, the user-manipulated graph is stored with both the function call label and the context-sensitive ID. By doing so, Kanon can uniquely identify the function call at runtime by using the call label and the context-sensitive ID. Additionally, because all call labels are maintained even with editing, Kanon can maintain the expected graphs in the same position and the same context even if the program is edited.

For the more intuitive user interface, we devised the implementation so that the programmer can build the structure without explicit changing edit mode such as "Add Node" and "Edit Edge". To enable the original manipulation feature of vis.js, the user has to click "Edit" button, positioned at the upper-right of the pop-up window in Figure 4.2. However, most software that the user directly manipulates displayed graphical representation by pointing devices does not require its wasteful action.

[7]This feature was not available when we carried out the user experiment in Chapter 3.

Based on the fact, we implemented the manipulation feature without clicking the "Edit" button. Concretely, holding down an edge invokes editing the reference of the edge. Double-clicking a node or an edge invokes editing the label of the target.

4.3.6.2 Automated Testing

After inputting an expected structure into a function call, Kanon automatically checks whether the actual structure after executing the function call matches the expected structure. On the code conversion, which is the step ① in Section 4.3.1, Kanon converts function calls as shown in Listing 4.3. in Listing 4.3, Kanon performs the following process:

- In the function call `func` at line 4, Kanon executes the original function call, and binds the returned object to a variable `retObj`. Kanon uses the variable `retObj` to check which object the function returns.
- In the function call `match` at line 9, Kanon checks whether the actual structure is the same as the expected structure. This function returns `true` if the runtime object structure is the same structure as the expected structure.
- If `match` returns `false` in the previous process, Kanon overrides the properties of the objects to be the same as the expected structure in a function `override` at line 10 (described in Section 4.3.6.3). After execution, binds the returned object to a variable `varRefs`, whose key is a variable name and whose value is an object the variable should refer to.
- Finally, Kanon changes variable references in a `for` statement written from line 12 to line 15 by using an object the function `override` returns.

LISTING 4.3: A simplified code conversion example of function call `func()`. We assume the variables `callLabel` and `contextSensitiveId` are already defined.

```
1  (() => {
2      var retObj, error, expectedGraph = ...;
3      try {
4          retObj = func(...);
5          error = false;
6      } catch (e) {
7          error = true;
8      } finally {
9          if (expectedGraph && (error || !match(objs, retObj,
               expectedGraph))) {
10             let varRefs = override(objs, retObj, expectedGraph);
11             let varNames = Object.keys(varRefs);
12             for (let i = 0; i < varNames.length; i++) {
13                 let obj = varRefs[varNames];
14                 eval(varNames[i] + " = obj");
15             }
16         }
17     }
18     return retObj;
19 })()
```

The function `match` checks the runtime objects as follows:

- Kanon traverses both graph simultaneously by depth first. The traversal starts from nodes referred by variables, therefore Kanon does not care about unreferable nodes.
- On each node, Kanon checks the consistency of properties, displayed label, type, and unique ID. Here, the properties mean the names of properties the focused node has. The displayed label means a text labeled to a node. The type means a type of a node at runtime such as `"string"`, `"number"`, and `"object"`. The unique ID is runtime object ID which is assigned at runtime to identify nodes.
- If matching, then Kanon traverses the next node. Otherwise, the function `match` returns `false`. When Kanon have traversed all nodes, then the function `match` returns `true`.

4.3.6.3 Object Overriding

After that, Kanon follows these procedures to override the properties of the objects in a function `override`.

- Kanon deletes all properties of all objects. At this time, all objects and their unique IDs are stored in order to be able to refer all objects. If the expected structure includes newly created nodes, Kanon constructs objects represented the nodes by using class constructors which are taken from an extra argument.
- Kanon lets all objects refer to other objects connected edges of the stored graph. When a type of the reference destination of an object is not `"object"` (i.e., the reference destination is literal), the literal is assigned to the property of the object.
- Referring to edges of stored expected graph, this function returns an object whose keys are variable names and whose values are objects the variable name will refer to. (The returned object is bound to the variable `varRefs` at line 10 in Listing 4.3.)

After overriding the objects' properties, Kanon needs to rebind variables to objects properly to be the same structure as the expected structure. This is because the variables are not accessible in the scope of the function `override`. In the `for` statement from line 12 to line 15, Kanon rebinds the variables to appropriate objects by using a function `eval`. Additionally, Kanon can change an object the original function returns, which is represented by `retObj`.

The new objects by overriding should be constructed by appropriate constructors. For example, if a programmer add one node labeled `Node` in the manipulation window of Figure 4.2, the object has to have methods defined in class `Node`. To construct an object by an appropriate class, the function `override` actually takes one more argument. The argument includes the constructors whose names are labels of nodes in the expected graph. The function `override` uses the constructors to construct new objects properly.

4.3.6.4 Reconstruct Expected Structure

The programmer's editing of the program affects the actual structure. Therefore, the programmer might expect a structure that is different from the one given before

editing. Here, we describe an automated mechanism to reconstruct new expected structure.

4.3.6.4.1 Why Do We Need Structure Reconstruction?

We describe a situation where we need structure reconstruction. We assume that the user program is shown in Figure 4.2, and the programmer has put the expected structure as shown in Figure 4.5 to the function call `reverse` at line 9. In this case, the user expects the built structure only when the structure just before executing the function call is the same structure as the structure of the window in Figure 4.2. More concretely, in the case that the programmer inserted an element to the middle of the list after building the expected structure shown in Figure 4.5, the programmer expects a structure where the new element has been inserted to Figure 4.5. Therefore, when the structure just before executing the function call changes, the expected structure should also be updated.

One solution in the situation is to remove the expected structure and let the programmer specify the new expected structure. However, the solution increases the burden because the programmer needs to build the newly expected structure.

4.3.6.4.2 Reconstruting New Expected Structure

When the expected structure was defined, the programmer operated the structure before executing the function call to put the expected structure. Here, we call a structure before executing a function call *preconditional structure*. Based on the fact, when the program is edited, it can be understood that Kanon can generate a plausible expected structure by applying the operations to a new preconditional structure.

Our choice to solve the problem is that Kanon automatically reconstructs new expected structure by reproducing the mouse operations. We explain the mechanism to reconstruct new expected structure.

As a preliminary preparation, we assume that Kanon has preserved the mouse operation when constructing the expected structure, and stored a preconditional structure just before executing a function call. When running the user program again, Kanon checks the actual structure before executing the function call is the same as the preconditional structure by using the function `match`. If it doesn't match, Kanon tries to reconstruct new expected structure. In the reconstructing step, Kanon reproduces the stored mouse operations, such as “add node", “change variable edge", and “delete edge". If all operations could be applied to the preconditional structure, Kanon deals with the applied structure as new expected structure.

Actually, however, all operations cannot always be applied because, for example, any nodes might be removed. If the reconstructing failed, instead of updating the expected structure, Kanon warns the programmer that please confirm the expected structure by making the function call parentheses become purple[8]. The programmer needs to confirm the warned structure to remove the warning.

[8] Additionally, it also makes the circle of the context, which contains the warned function call, on the call tree become purple.

4.4 Case Studies

4.4.1 Reversing a Linked List

Specification: the `reverse` method on linked lists re-links the elements in the reverse order, returning the head element of the reversed list.

First, we wrote the program halfway as shown in Figure 4.2. At the time, we expected the structure after executing the recursive call as shown in Figure 4.5, and built and registered the structure. After that, Kanon shows the graph as shown in Figure 4.5 in the visualization pane.

Programming experiences with and without the proposed features differ in the following ways. Since the existing Kanon shows the same object graph structure before/after the function call, we needed to imagine the structure after the recursive call. However, since Kanon with the features shows the object graph reflecting the expected structure, the programmer can write the next statement `this.next.next = this;` after the recursive call by seeing the visualization.

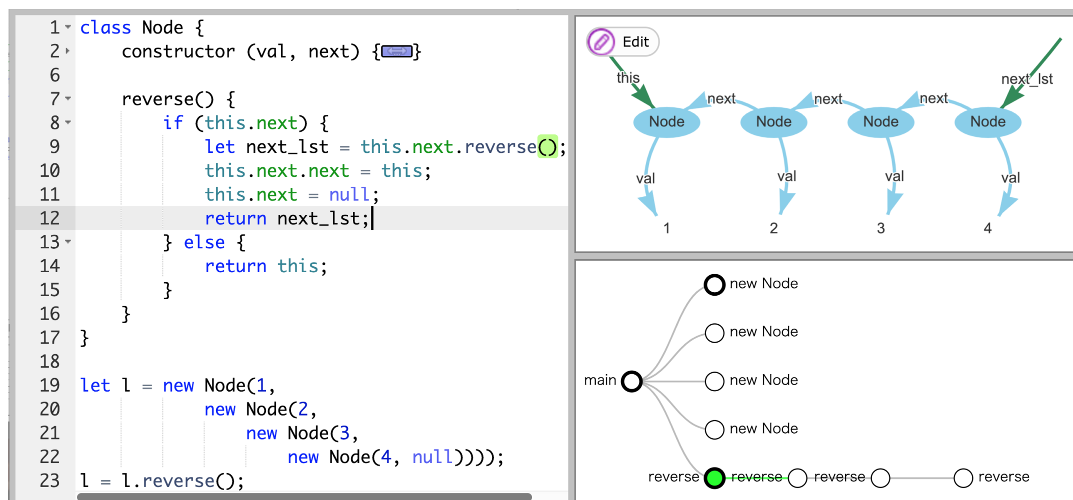


FIGURE 4.10: A screenshot of Kanon after the programmer has completed defining the recursive function `reverse`.

Figure 4.10 shows a scene when we completed the recursive function definition. Since the parentheses of the recursive call and the node of the call tree in Figure 4.10 become green, we immediately understood that the definition is correct (at least with respect to the test case).

4.4.2 Insertion Sorting of a Linked List

Specification: the `insertSort` method of a linked list destructively reorders the elements of the list in the order of their 'value's.

We completed the definition in the following steps. First, we created a list (lines 19–23) and a call to `insertSort` (line 24). We added an expected structure (i.e., a sorted list) to this call. Second, we defined an empty `insertSort` method of `Node` (lines 7,14) and inserted a conditional branch with empty then/else branches (line 8). Third, we inserted a recursive call to `insertSort` on `next` in the then-branch (line 9), and added an expected structure (i.e., a sorted list except for the first element). Fourth, we realized that we need to insert `this` element into the sorted list by looking at the visualization of the expected structure. So we inserted a call to a helper function `insert` on the sorted list (line 11) with its empty definition (line 16). We added an

expected structure after `insert`, which turns the first test case (on line 24) succeeded. Figure 4.11 shows the screenshot at this point.

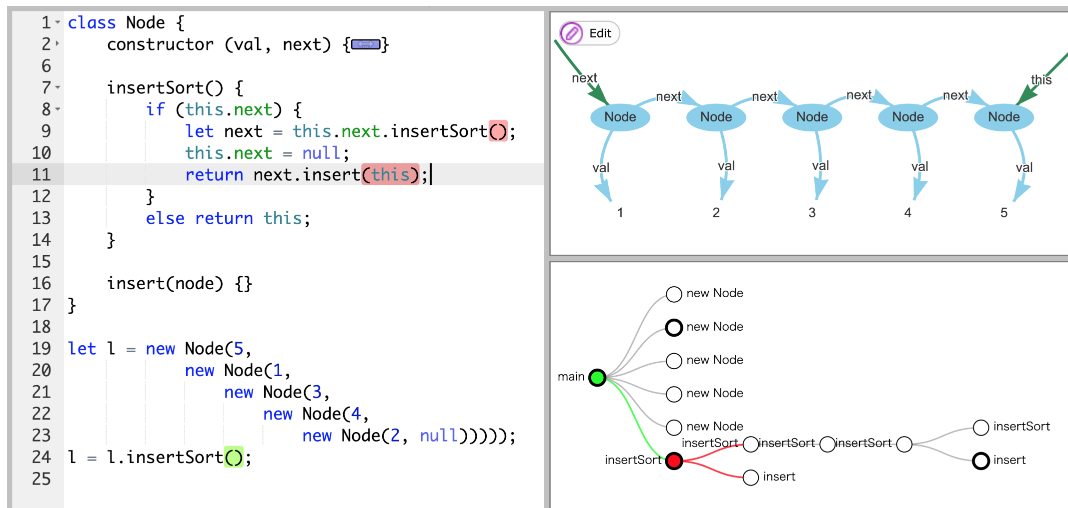


FIGURE 4.11: A screenshot after writing the body of `insertSort`.

Interestingly, the call tree in Figure 4.11 shows only two calls to `insert`. This is because the incomplete definition of `insertSort` returns empty results, which eventually causes errors upon subsequent call to `insert`. After added two more expected structures to the calls to `insertSort`, the call tree becomes like Figure 4.12. It was easy to add them since we only needed to modify part of the list.

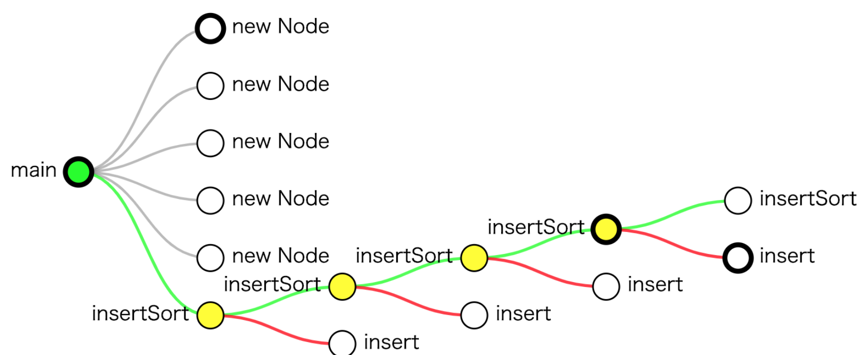


FIGURE 4.12: A call tree after the programmer has inserted expected structures to the call, written at line 9 and line 11 on Figure 4.11, on all contexts.

Finally, we wrote the body of `insert` by selecting one of four registered test cases in turn. The final screenshot is Figure 4.13. Since we incrementally added conditional branches as a test case required it, the resulting body was rather redundant. Nevertheless, the four test cases made us more confident in the process of definition, and also made us feel easier to refactor the redundant conditional branches.

4.4.3 Implementing Interpreter for Lambda Calculus of Tree Structure

Specification: the `eval` method of a lambda-term (either of `App`, `Lam`, or `Var` class) evaluates itself into a normal form.

```
 7    insertSort() {…}
15
16    insert(node) {
17        if (this.next) {
18            if (this.val > node.val) {
19                node.next = this;
20                return node;
21            } else {
22                this.next = this.next.insert(node);
23                return this;
24            }
25        } else {
26            if (this.val > node.val) {
27                node.next = this;
28                return node;
29            } else {
30                this.next = node;
31                return this;
32            }
33        }
34    }
35 }
```

FIGURE 4.13: A screenshot when the `insert` method succeeded in all the tests.

We made the following three top-level test inputs, namely $\langle\langle(\lambda x.x)\ (\lambda y.y)\ z\rangle\rangle$`.eval()`, $\langle\langle(\lambda x.(\lambda y.y\ x))\ z\ w\rangle\rangle$`.eval()`, and $\langle\langle(\lambda x.(\lambda x.x))\ (a\ b\ c)\rangle\rangle$`.eval()`[9]. We also provided the expected results to those test cases. Then, we implemented the `eval` and auxiliary methods in a breadth-first order; we took one unimplemented method call with a test case, wrote its body. When there was a further method call without a proper implementation, we just add an expected structure and continue the implementation of the caller's body. Here, for simplicity, we assume that there is no input that causes free variable capturing during the evaluation.

Figure 4.14 shows a screenshot in the middle of implementation, which successfully evaluates $\langle\langle(\lambda x.x)\ (\lambda y.y)\ z\rangle\rangle$`.eval()`. In this case, we needed to add 8 expected structures by following the abovementioned implementation order. When we define the `eval` method $\langle\langle(\lambda x.x)\ (\lambda y.y)\ z\rangle\rangle$`.eval()`, we wrote 3 method invocations. After writing the first method invocation (i.e., $\langle\langle(\lambda x.x)\ (\lambda y.y)\rangle\rangle$`.eval()`) at line 5, without the proposed feature, we would have to switch the context to the callee, leaving the incomplete implementation aside. With our proposed feature, however, we were able to continue on the caller's side (i.e., lines 6 and 7) with visualized objects. When writing line 7, we were able to see the structure of `t1` and `t2` as visualized objects of $\langle\langle\lambda y.y\rangle\rangle$ and $\langle\langle z\rangle\rangle$, respectively.

As a result, we can complete the definition of the `eval` methods, including other necessary methods, by building the expected structures upon 26 method invocations.

When defining a program such as an interpreter, we need to define evaluation functions and auxiliary functions every kind of expressions and its combination. Therefore, in the case of test-driven development using unit test, we need to build a lot of small test cases. However, in this case, we completed the implementation by only creating three lambda expressions as the top-level inputs. This is because, for example, defining the body of $\langle\langle(\lambda x.x)\ (\lambda y.y)\ z\rangle\rangle$`.eval()` derives test executions such as $\langle\langle z\rangle\rangle$`.eval()` and $\langle\langle\lambda y.y\rangle\rangle$`.apply(`$\langle\langle z\rangle\rangle$`)`.

[9] We assume that $\langle\langle t\rangle\rangle$ is an object that denotes lambda-term.

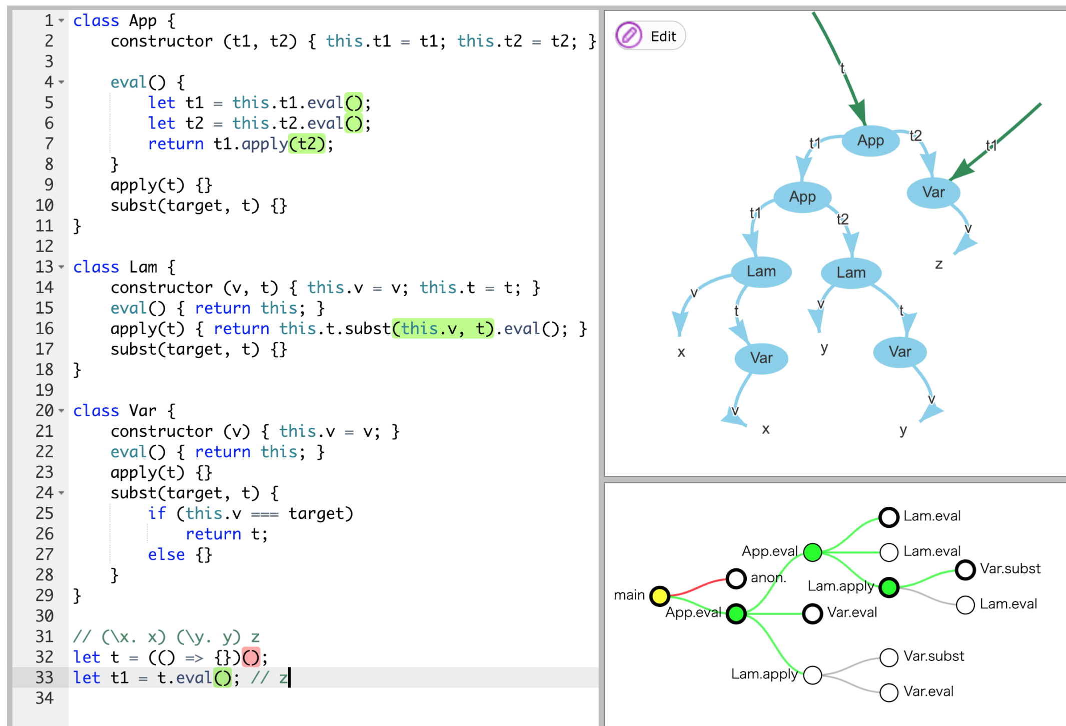


FIGURE 4.14: A screenshot after we use just one test case to define the methods of `eval`, `apply`, and `subst`.

4.4.4 Result

Throughout the case studies, we observed the following benefits and issues:

- Benefit: We felt that we were less interrupted during the implementation. In other words, we did not become a state that we do not know what to write. This is probably because the top-down implementation order requires less context switch.
- Benefit: We were able to notice mistakes earlier thanks to the automated testing. Some of the mistakes we made actually failed test cases provided for the other calling contexts than the currently implementing one. We were able to notice them immediately.
- Issue: It was not easy to find missing cases when the implementation is not yet complete. Though it is not a particular issue of Kanon but common to the test-driven development, some mechanisms to guide exhaustiveness would be helpful.
- Issue: The visualization is sometimes cluttered with *dead* objects, especially with the lambda-calculus interpreter. A garbage collection mechanism for visualized objects would be useful for pure functional programming.

4.5 Related Work

EG is an Eclipse plug-in for example-centric programming (Edwards, 2004), which has a feature to attach an *assertion* or an *override* to an expression under a specific execution context. This feature is similar to our proposed features to add an expected

structure to a function call under a specific execution context, but different in (1) that EG only supports primitive values whereas we focus on data structures, and (2) that EG treats assertions and overrides separately whereas our expected structures are used for overriding as well as testing. Interestingly, the EG paper (Edwards, 2004) does not explicitly discuss need for the overriding feature at defining recursive functions, even though its running example is a recursive factorial function.

Kanon assumes that the programmer writes a program in a test-driven development (Beck, 2003) style. Such a style can be commonly found in live programming (Victor, 2012; Imai, Masuhara, and Aotani, 2015) as well as in example-centric programming (Edwards, 2004).

Vital is an interactive graphical environment for Haskell (Hanna, 2002), which visualizes data structures including recursive data structures and lazily evaluated infinite lists. It does not provide the automated testing/overriding features, i.e., it only visualizes data structures in an actual execution of the source code. It supports a direct manipulation mechanism, in which manipulation of visualized values by using a pointing device will change the user program so that it will generate such manipulated values.

Chapter 5

Conclusion

This dissertation reports two studies on Kanon: The one is a user experiment for the initial version of Kanon, the other is development of a feature that supports recursive functions.

We carried out qualitative user experiment to collect programmers' opinions about the implementation of existing Kanon. Through observation and the interview, it was found that, though most of the participants had positive impressions, Kanon still has room for improvement. Also, we observed interesting programming behavior with Kanon. When errors occurred, many participants took, when using Kanon, longer time to notice occurrence of errors partly because of the design of the existing Kanon. For the `reverse` task, several participants with Kanon took an inappropriate strategy to solve the problem.

We propose a set of features for defining recursive functions in a live data-structure programming environment Kanon as a solution of one of the problems found in the experiment. With the features, the programmer can enjoy live programming experience by merely building an expected structure upon a call to an incomplete recursive function. The expected structure also serves as an automated test case, guiding the programmer where to write next. We confirmed, through case studies, that the proposed features can work well when defining typical recursive functions. The features are implemented as an extension to Kanon, and are publicly availabe at https://github.com/prg-titech/Kanon.

5.1 Future Work

The feature that this dissertation proposed has room for improvement. In the current implementation, Kanon reconstructs the expected structure when the preconditional structure is changed. However, the reconstructing is only a temporary solution. Kanon should be able to provide users with alternative ways such as partially testing, in which the user can partially specify expected parts of the structure. Additionally, a user study should be carried out to further evaluate the usability of this feature.

Kanon still has more future improvements found through the experiment.

One of the future work is to support loops. In this dissertation, the author proposed a set of features to support recursive function. However, depending on the type of data structure, it may be easier to define the structure by loops rather than recursion. More than half the participants of the experiment actually used loops to define the `reverse` method that reverses a doubly-linked list. When they wrote a code fragment in the loop body by focusing on a certain context, an error occurred in a context other than the focused context. Therefore, Kanon needs a feature to make it easier for users to define loops, for example, synthesizing the body of the loop by letting users input a specification (or behavior) of each context of the loop.

Second, Kanon should present a more improved error report. Though it was found, through the experiment, that notifying the programmer and error as early as possible is crucial, the problem is not trivial in a live programming environment because an incomplete program that the environment treats frequently becomes an erroneous state. Therefore, the live programming environment needs to present an appropriate error report not to disturb their programming.

Third, a direct manipulation mechanism is one of the future work of Kanon. This is based on the fact that, though the visualization of the snapshot view mode helps a programmer to understand a structure after an execution reaches the cursor position, the programmer has to convert an operation that he or she considered by seeing the graph into a statement in their mind. The direct manipulation mechanism of Kanon allows programmers to define methods just by manipulating the node-link diagram directly.

Fourth, the automatic layout engine should be more improved. The current automatic layout engine implemented in Kanon specifies special layout behavior for some structures, which currently include binary trees and linked list. However, there is no way to know what kind of structure the user assumes. Therefore, Kanon should introduce a structure-aware layout algorithm to lay out arbitrary structures.

Finally, Kanon still has room for improvement of feedback performance. Kanon executes a converted program every the user program is executable. In the case studies in Section 4.4, the feedback Kanon provided is *live*. However, when the length of the list is around 15, the feedback Kanon provided is not *live*[1].

[1]The feedback under 50 ms is common to live programming environments (Rein et al., 2016).

Bibliography

Aaron, Samuel and Alan F. Blackwell (2013). "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages". In: *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*. FARM '13. Boston, Massachusetts, USA: ACM, pp. 35–46. ISBN: 978-1-4503-2386-4. DOI: 10.1145/2505341.2505346. URL: http://doi.acm.org/10.1145/2505341.2505346.

Apple Computer (2016). *Swift Playgrounds*. http://www.apple.com/swift/playgrounds/. Accessed February 2017.

Archambault, D. and H. C. Purchase (2012). "The Mental Map and Memorability in Dynamic Graphs". In: *2012 IEEE Pacific Visualization Symposium*, pp. 89–96. DOI: 10.1109/PacificVis.2012.6183578.

Beck, Kent (2003). *Test-Driven Development: by Example*. Addison-Wesley Professional.

Brown, M. H. (1991). "Zeus: a system for algorithm animation and multi-view editing". In: *Proceedings 1991 IEEE Workshop on Visual Languages*, pp. 4–9. DOI: 10.1109/WVL.1991.238857.

Brown, Marc H. and Robert Sedgewick (1984). "A System for Algorithm Animation". In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '84. New York, NY, USA: ACM, pp. 177–186. ISBN: 0-89791-138-5. DOI: 10.1145/800031.808596. URL: http://doi.acm.org/10.1145/800031.808596.

Burckhardt, Sebastian et al. (2013). "It's Alive! Continuous Feedback in UI Programming". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, pp. 95–104. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462170. URL: http://doi.acm.org/10.1145/2491956.2462170.

B.V., Almende (2018). *vis.js - A dynamic, browser based visualization library*. URL: http://visjs.org/.

Edwards, Jonathan (2004). "Example Centric Programming". In: *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*. Ed. by Doug Schmidt. Vancouver, BC, Canada: ACM, pp. 84–91. ISBN: 1-58113-831-9. DOI: 10.1145/1052883.1052894. URL: http://doi.acm.org/10.1145/1052883.1052894.

Goldman, Max, Greg Little, and Robert C. Miller (2011). "Real-Time Collaborative Coding in a Web IDE". In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. Santa Barbara, California, USA: ACM, pp. 155–164. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047215. URL: http://doi.acm.org/10.1145/2047196.2047215.

Granger, Chris (2012). *Light Table*. http://www.chris-granger.com/lighttable/. Accessed February 2017.

Gulwani, Sumit (2010). “Dimensions in Program Synthesis”. In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. PPDP ’10. Hagenberg, Austria: ACM, pp. 13–24. ISBN: 978-1-4503-0132-9. DOI: 10.1145/1836089.1836091. URL: http://doi.acm.org/10.1145/1836089.1836091.

Guo, Philip J (2013). “Online Python Tutor: Embeddable Web-based Program Visualization for CS Education”. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM, pp. 579–584.

Hancock, Christopher Michael (2003). “Real-time Programming and the Big Ideas of Computational Literacy”. PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology.

Hanna, Keith (2002). “Interactive Visual Functional Programming”. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP ’02. Pittsburgh, PA, USA: ACM, pp. 145–156. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581493. URL: http://doi.acm.org/10.1145/581478.581493.

Hendrix, T. Dean, James H. Cross II, and Larry A. Barowski (2004). “An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE”. In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’04. Norfolk, Virginia, USA: ACM, pp. 387–391. ISBN: 1-58113-798-2. DOI: 10.1145/971300.971433. URL: http://doi.acm.org/10.1145/971300.971433.

Hidayat, Ariya (2018). *Esprima*. URL: http://esprima.org.

Imai, Tomoki, Hidehiko Masuhara, and Tomoyuki Aotani (2015). “Making Live Programming Practical by Bridging the Gap Between Trial-and-error Development and Unit Testing”. In: *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. Ed. by Jonathan Aldrich. ACM. Pittsburgh, USA: ACM, pp. 11–12. ISBN: 978-1-4503-3722-9. DOI: 10.1145/2814189.2814193.

Jakobs, Fabian (2018). *Ace - The High Performance Code Editor for the Web*. URL: https://ace.c9.io.

Kato, Jun, Sean McDirmid, and Xiang Cao (2012). “DejaVu: Integrated Support for Developing Interactive Camera-based Programs”. In: *Proceedings of the 25th annual ACM symposium on User Interface Software and Technology (UIST’12)*. ACM, pp. 189–196.

Khan Academy (2018). *Intro to JS: Drawing & Animation*. https://www.khanacademy.org/computing/computer-programming/programming. Accessed February 2017.

Lee, Yi-Yi, Chun-Cheng Lin, and Hsu-Chun Yen (2006). “Mental Map Preserving Graph Drawing Using Simulated Annealing”. In: *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation - Volume 60*. APVis ’06. Tokyo, Japan: Australian Computer Society, Inc., pp. 179–188. ISBN: 1-920682-41-4. URL: http://dl.acm.org/citation.cfm?id=1151903.1151930.

Lieberman, Henry and Christopher Fry (1995). “Bridging the Gulf Between Code and Behavior in Programming”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’95. Denver, Colorado, USA: ACM Press/Addison-Wesley Publishing Co., pp. 480–486. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223969. URL: http://dx.doi.org/10.1145/223904.223969.

McDirmid, Sean (2007). “Living it Up with a Live Programming Language”. In: *In Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* ACM, pp. 623–638.

— (2013). "Usable Live Programming". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, pp. 53–62. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509585. URL: http://doi.acm.org/10.1145/2509578.2509585.

Oka, Akio, Hidehiko Masuhara, and Tomoyuki Aotani (2017). "The Visualization of Data Structures and Interactive Features for Live Programming". In: *The 113th IPSJ Workshop on Programming*. Japanese.

— (2018). "Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Boston, MA, USA: ACM, pp. 72–87. ISBN: 978-1-4503-6031-9. DOI: 10.1145/3276954.3276962. URL: http://doi.acm.org/10.1145/3276954.3276962.

Oka, Akio et al. (2017a). "Kanon: Data Structure Programming using Live Programming Environment". In: *Proceedings of the 10th JSSST Workshop on Programming and Programming Languages*. PPL 2017. Japanese.

— (2017b). "Live Data Structure Programming". In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. Programming '17. Brussels, Belgium: ACM, 26:1–26:7. ISBN: 978-1-4503-4836-2. DOI: 10.1145/3079368.3079400. URL: http://doi.acm.org/10.1145/3079368.3079400.

Prototype Window Class. https://github.com/sgruhier/prototype_window. Accessed January 2019.

Rein, Patrick et al. (2016). "How Live Are Live Programming Systems?: Benchmarking the Response Times of Live Programming Environments". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. PX/16. Rome, Italy: ACM, pp. 1–8. ISBN: 978-1-4503-4776-1. DOI: 10.1145/2984380.2984381. URL: http://doi.acm.org/10.1145/2984380.2984381.

Resig, John (2012). *Redefining the Introduction to Computer Science*. http://ejohn.org/blog/introducing-khan-cs/. Accessed March 2018.

Stasko, John T (1989). *TANGO: A Framework and System for Algorithm Animation*. Tech. rep. Providence, RI, USA.

Teitelman, W. and L. Masinter (1981). "The Interlisp Programming Environment". In: *Computer* 14.4, pp. 25–33. ISSN: 0018-9162. DOI: 10.1109/C-M.1981.220410. URL: http://dx.doi.org/10.1109/C-M.1981.220410.

Victor, Bret (2012). *Inventing on Principle*. Keynote Talk at the Canadian University Software Engineering Conference (CUSEC). Montreal, Quebec.