Live Data Structure Programming

Akio Oka
School of Computing
Tokyo Institute of Technology
Tokyo, Japan
a.oka@prg.is.titech.ac.jp

Hidehiko Masuhara
School of Computing
Tokyo Institute of Technology
Tokyo, Japan
masuhara@acm.org

Tomoki Imai
School of Computing
Tokyo Institute of Technology
Tokyo, Japan
tomo832@gmail.com

Tomoyuki Aotani
School of Computing
Tokyo Institute of Technology
Tokyo, Japan
aotani@is.titech.ac.jp

ABSTRACT

When we write a program that manipulates data structures, we often draw and manipulate a visual image of the structures in our mind. In order to immediately connect those mental images with the data objects created by the program, we propose a live programming environment, called Kanon, specialized for data structure programming. It automatically executes a program being edited, and draws objects and their mutual references as a node-link diagram. In order to visualize information relevant to the programmer's concern, it offers two visualization modes based on the cursor position in the editor. It also offers two interactive mechanisms that relate elements in the program to elements on the diagram, and vice versa. The implementation includes a novel technique for mental map preservation of visual diagrams.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**;

KEYWORDS

Live programming, data structures, object graph

ACM Reference format:
Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. 2017. Live Data Structure Programming. In *Proceedings of Programming Experience Workshop, Brussels, Belgium, April 4, 2017 (PX/17),* 7 pages.
DOI: 10.1145/3079368.3079400

1 INTRODUCTION

Live programming [7, 16] is helpful to the programmer by giving "immediate connection" from a program to its execution result without letting the programmer run the program in his or her mind. Most of past demonstrations of live programming target programs whose results are not obvious from their texts, including the programs for drawing pictures [16], for synthesizing music [1], for animating game characters [14], and for teaching algorithms [11].

Data structure programs fall in the same category, and hence we believe live programming can be helpful 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.

We propose a live data structure programming environment that helps immediate connection between the program text and graphical images of data structures in the programmer's mind. Though the programmers could have variety of mental images for data structures, we assume that images with boxes and arrows are common enough. Figure 1 is an example of such an image for a doubly-linked list.

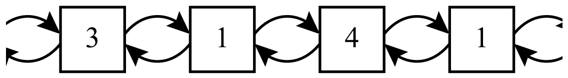


Figure 1: A mental image of a doubly-linked list.

While it is a straightforward idea and there is 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 [13], jGRASP [8] and Python Tutor [6], that visualize user-defined data structures, they mainly focus on the situation when the developer tries to examine behavior of programs in a *post-mortem* fashion. In other words, development and examination are separated processes in those environments.

In order to provide more live experiences with data structure programming, we need to consider visualization while the developer is writing and changing a fragment of code[1]. In the following sections, we discuss the characteristics of data structure programming

[1]While we primarily focus on the activities of writing and modifying code for data structures, we do not exclude the situations of debugging and code-understanding. When writing a new code fragment often involves with identifying problems in the written code, and understanding the existing code related to the one being written. This would be especially true with live programming.

Listing 1: Insertion of a node into the middle of a doubly-linked list

```
// insert the given value after the node with the given key
function insertAfter(dlist, key, value) {
  var cursor = dlist.head; //the first node of the list
  ...(move cursor to the node with key)...
  var temp = new Node(value);
  // insert temp after cursor
  temp.prev      = cursor;
  temp.next      = cursor.next;
  cursor.next.prev = temp;
  cursor.next      = temp;
}
```

(Section 2) and the design space of live date structure programming (Section 3), followed by the design our proposed programming environment, Kanon (Section 4) and its implementation issues (Section 5).

2 CHARACTERISTICS OF DATA STRUCTURE PROGRAMMING

Before discussing features of live programming, we here discuss characteristics of data structure programming, compared to other kinds of programming.

2.1 Difficulties with Textual Representations

While textual representations are the primary means of presenting data structures, they are not as easy as textual representation of other basic data types. For example, a textual representation of a list with 3, 1, 4 and 1 would be like this.

```
Node(val=3,
     next=Node(val=1,
               next=Node(val=4,
                         next=Node(val=1,next=Nil))))
```

This representation is verbose as it shows information at the field level[2].

When there are circular or shared references, textual representations become much more verbose. For example, a textual representation[3] of a doubly-linked list would become like this.

```
#1=Node(val=3, prev=Nil,
    next=#2=Node(val=1, prev=#1#,
              next=#3=Node(val=4, prev=#2#,
                        next=Node(val=1, prev=#3#,
                                  next=Nil))))
```

2.2 Manipulation of References

Data structure programming often involves with manipulation of references. For example, the bottom-half of the JavaScript function in Listing 1 inserts a node in the middle of a doubly-linked list. As we see, we need to manipulate forward and backward references of multiple objects, which can easily be mistaken.

[2]We here use a linked-list as an example of user-defined data structures. Though one might think that many languages can print lists in a more concise format, it is not the case for user-defined data structures in general.

[3]We here use a Common Lisp like notation with sharp symbols for showing shared references.

3 DESIGN PARAMETERS

There are many parameters to consider when we design a live programming environment for data structures. We here discuss those parameters based on the design of existing live programming environments. We will discuss our actual design decisions in the next section.

3.1 How Programs Should be Written

There are two types of live programming, namely *immediate execution upon modification* and *modification of running programs*. The former type just executes a program whenever a piece of the program is modified. The runtime of a program is expected to be short, so that the result can be displayed immediately. The latter is realized by a language runtime that allows hot-swapping, where a part of running program can be replaced without termination. This type is useful for long-running and interactive programs.

With both types, a program needs *top-level expressions* (i.e., the main function) that specify entry points and initial parameters of a program.

3.2 How Data Structures Should be Visualized

In order to construct immediate connection to programmer's mental images, visual representation should be intuitive. Existing live programming environments mainly use textual representation unless a program is written to draw visual images.

For data structures, a node-link diagram like the one shown in Figure 1 would be a suitable representation. Although there are many possibilities, it is common to draw one graphical element for each data element (object), and to draw an arrow from one graphical element to another when the former data element has a reference to the latter. For example, UML object diagram is a standardized node-link diagram.

Customized visual representation can be more efficient with respect to space and cognitive load. When we are using a list, we only concern the order of stored elements in most cases. Low-level information such as links between cons-cells would be too detailed and puts additional cognitive burden to users.

3.3 Which Data Should be Visualized

Live programming environments should choose an appropriate set of data elements to be displayed. Since too much displayed information could increase cognitive load of the programmer, some environments let the programmer to choose elements, and some other environments display elements that can fit within a programming editor window. For example, YinYang merely displays values of expressions that have *probes* attached by the programmer [14]. Swift Playground displays one value for each program line.

For data structures, selection of visualization target can be more important since their graphical representations consume larger area on the screen.

3.4 How Changes Should be Visualized

When a live programming environment visualizes data, the data can be changing during a program execution. For example, given a program that repeatedly approximates a mathematical function, we want to see the changes of intermediate results during a run. Existing environments can show such changes as a series of values [9, 14] or as a line chart [3]. For programs that produce visual images (i.e., drawing programs), there are attempts to use a stroboscopic visualization [5] or a timeline visualization [10].

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

3.5 How to Preserve a Mental Map when Program Changes

Live programming environments should *preserve the mental map* when a program is modified. Here, the mental map is correspondences between a visual representation and data elements in the programmer's mind. When a program is modified, the data elements produced by a new version of the program are different from the ones produced by the old version. However, in many cases, especially in live programming, a small modification to a program will change only a small part of the data elements produced by a program.

For example, adjusting constant parameters in a drawing program is one of 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 [11].

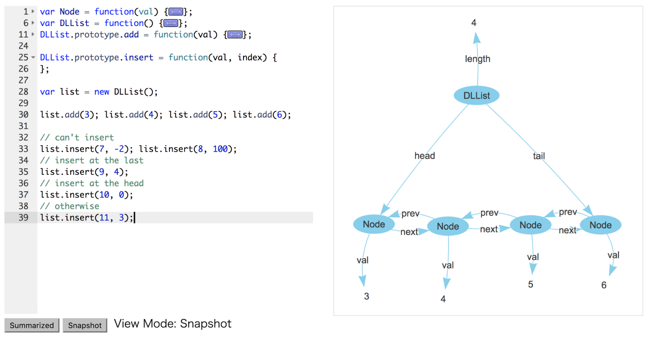
For data structures, preservation of the mental map is not a trivial task. Assume there is a program that creates two node objects; e.g., "`new Node(1); new Node(2)`", which are visualized as two graphical elements. When the programmer inserts "`new Node(3)`" in between the two `new` expressions, one more graphical element shall be added. Since the programmer considers the second `new` expression in the modified program is newly added, this should be drawn as an additional graphical element to preserve the mental map. However, by comparing two versions of program texts, it is not obvious whether the second `new` expression in the new version is newly added, or is a result of a parameter change. The problem can be more complicated when a modification of a method call indirectly causes changes of object creation, for example.

3.6 How to Relate Visualized and Program Elements

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

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 establishing connection between those visual elements and program elements.



```
var Node = function(val) {↔};
var DLList = function() {↔};
DLList.prototype.add = function(val) {↔};

DLList.prototype.insert = function(val, index) {
};

var list = new DLList();

list.add(3); list.add(4); list.add(5); list.add(6);

// can't insert
list.insert(7, -2); list.insert(8, 100);
// insert at the last
list.insert(9, 4);
// insert at the head
list.insert(10, 0);
// otherwise
list.insert(11, 3);
```

Figure 2: A Screenshot of Kanon.

4 KANON: A LIVE DATA STRUCTURE PROGRAMMING ENVIRONMENT

We designed a JavaScript-based live programming environment, Kanon, for data structures. Here, we first give an overview of the environment followed by the design parameters discussed in the previous section. We then explain specific features in detail.

4.1 Overview

Figure 2 is a screenshot of Kanon. The left- and right-hand sides are the editor pane where a program is written in, and a visualization pane that displays data structures, respectively. For simplicity reasons, it currently supports single program file. The editor has a syntax highlighting feature (including showing syntax errors). Each circle in the visualization pane represents an object that is created during an execution. The label on the circle is the class name of the object. Arrows from a circle show the field values in the object, which either point to other objects or primitive values.

4.2 Design Decisions

How Programs Should be Written. We take the *immediate execution upon modification* style. This is because we mainly consider definitions of data structures and methods that manipulate those structures. With an adequately modularized design, execution of each method should be short-lived, and can easily be driven by unit test cases.

In our current implementation, a program is merely executed from top to bottom. Therefore, test cases should be written after the definitions of the data structures.

How Data Structures Should be Visualized. As we can see in Figure 2, we draw an oval and an arrow for an object and a value in a field, respectively. The labels on an oval and on an arrow show the class and field names, respectively.

We currently use an automated graph layout algorithm available in a visualization toolkit (vis.js) so as to arrange ovals with some distance, and without much overlapping. Current implementation does not use information such as classes and object ownerships, which should certainly be improved in future as the generated layouts often require manual rearrangements. (In fact, we manually rearranged objects in Figure 2.)

For the ease of implementation, an array of n elements is visualized as an object that has n fields. As this is certainly awkward, future implementation will consider alternative representations.

Which Data Should be Visualized. We basically show all objects created in a run of a program. The visualization is implicit, i.e., the programmer does not need to specify objects to be displayed.

While it could lead to a situation where the visualization is flooded by too many objects, it is convenient for the programmer to write small definitions. With the notion of context (which is explained later), it would also be easy to introduce a mechanism to filter objects created in a specific test case.

How Changes Should be Visualized. We provide two ways of showing changes in a program run: the one is to animate the graphical representation by using cursor movement in the text editor, and the other is to show summarized effects of changes in one graphical representation. Both are explained in Section 4.3.

How to Preserve a Mental Map when Program Changes. As for changes caused by program modification, we proposed an implementation technique to cope with a technical problem, namely the mental map preservation. We discuss this issue in Section 5.2.

How to Relate Visualized and Program Elements. We provide two mechanisms that help connecting program elements to visual elements and vice versa. The first mechanism is a *probe* that displays a value of a variable on the graphical representation. The second mechanism is called *jump-to-contruction*, which is invoked by a click on a graphical element, and moves the cursor position to the program element that corresponds to the graphical element. Those two mechanisms are explained in Section 4.4.

4.3 Snapshot and Summarized View Modes

Kanon provides two view modes for object graphs, namely the *snapshot view mode* and the *summarized view mode.*

With the *snapshot view mode*, the view shows the object graph when the program execution reached at 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* is chosen[4].

Figure 3 is an example of a snapshot view for the program text in Listing 2 (in which the cursor position is denoted by a black rectangle), in the context of `list.add(4)`. 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 at 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 command (such as assignment to an object field and construction of an object) at the cursor position on top of the object graph at the end of the execution.

[4]The current implementation does not show the function calls that form the context, which is left for future work. Drawing a segmented line that connects the call sites as done in Dr. Racket (https://racket-lang.org/) is one possibility.

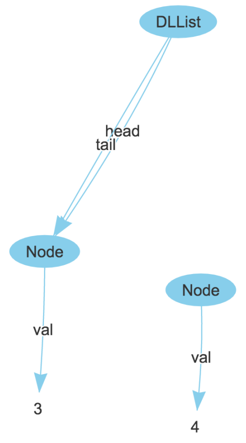


Figure 3: A snapshot view in the context of `l.add(4)`.

Listing 2: Partially defined add.

```
// add the given val at the end of the list
DLList.prototype.add = function(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);
```

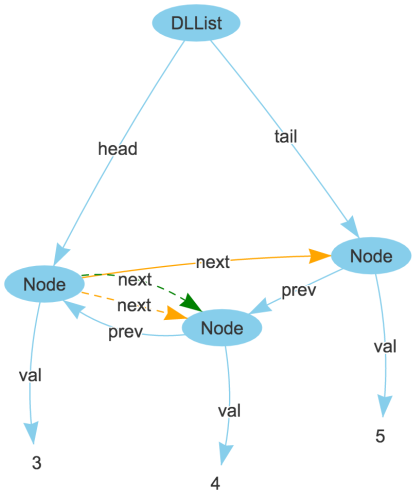


Figure 4: A summarized view for the program in Listing 3.

Listing 3: Finished (yet incorrect) definition of add.

```
// add the given val at the end of the list
DLList.prototype.add = function(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 4 is an example of a summarized view for the program text in Listing 3, where the programmer 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 shows the reference found at the end of execution. The dash arrows denote the overwritten references, i.e., once created by this (orange) or other (green) assignment, and then disappeared due to later assignments.

From the diagram, we can observe that the final graph is not right. The next field of the leftmost Node should reference the middle Node, whereas it references the rightmost Node in the final state. From the view, we can also see that the reference was initially correct (as shown with the green dashed arrow) 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.

4.4 Bidirectional Connection between Code and Graphical View

Kanon offers two mechanisms, namely a *probe* and the *jump-to-construction* mechanism, that help the programmer to grasp the connection between code elements and graphical elements.

A *probe*, which is a $ symbol attached at the beginning of a variable declaration, adds the variable name in the graphical view with an arrow pointing to the object that the variable references to.

For example, when we add a $ symbol to temp in Listing 2 (i.e., changing the second line to “var $temp = new Node(val);”), the view will have an arrow that points to the value in the variable.

A click on a node or an edge in the graphical view invokes the *jump-to-construction* mechanism, which moves the cursor to the command in the program that created the clicked graph element (either a new expression or a field-assignment statement).

5 IMPLEMENTATION

We implemented a prototype of Kanon for JavaScript running on web browsers. It is available online[5]. Below, we first overview the

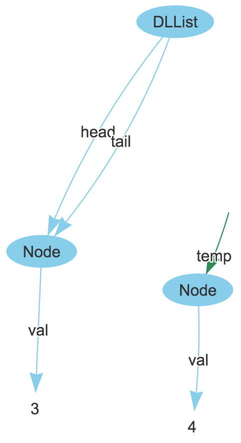


Figure 5: A snapshot view with a probe.

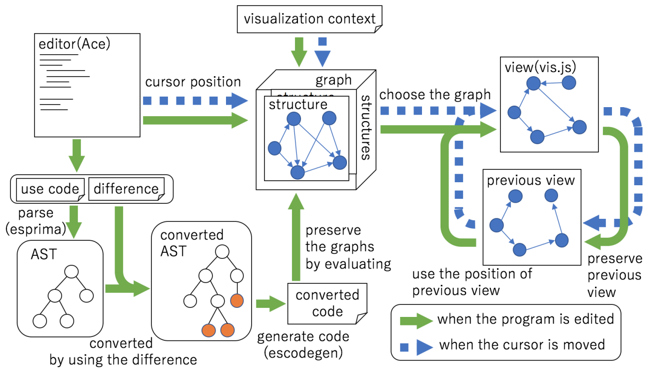


Figure 6: Overview of the Implementation

implementation and then describe our novel technique to preserve mental map.

5.1 Overview

Figure 6 overviews the implementation. We explain the structure by following the operations taken place upon a program modification.

We use a modified version of the Ace editor[6] for editing a program text. When the programmer edits a piece of text, it notifies the visualization engine.

The visualization engine parses the program text in the editor by using Esprima[7]. 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.
- For each new expression, it appends a piece of code that records object ID in a special field of the created object.
- For each new and field assignment expression, it inserts checkpointing code before and after the expression. The

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

[6]https://ace.c9.io/

[7]http://esprima.org/

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.
- For a variable declaration with `$` (a symbol for *probing*), it removes the symbol from the variable name, and records the variable name in the traversal environment.

The engine then transforms the syntax tree back to a program text, and evaluates it by using `eval`. The checkpointing code in the program, when executed, records JavaScript objects that can be reachable from the provided objects. We use the object reflection mechanism to obtain field values from an object. The objects and their references are recorded as a graph data (i.e., nodes and links) with a virtual timestamp that increases every checkpointing execution.

To update a 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 that corresponds to the current visualization context. Finally, it selects the object graph that is recorded at the nearest checkpoint within the calculated timestamp range.

The object graph is visualized by using the vis.js visualization library[8]. It specifies geometric locations of object nodes by using the locations of the currently displayed graph. When there are objects that do not exist in the current graph, their locations are automatically determined by using the physics layout algorithm is vis.js.

5.2 Mental Map Preservation

Kanon updates the graphical view without breaking the *mental map*. Here we first explain the requirements, and then describe the mechanism for the preservation.

When a program is modified, the new program may construct a different object graph from the previous one. In most cases, the modification in the program merely makes partial change in the object graph (for example, addition of a new object, and change of references between objects). Hence the programmer expects that the visual change is also partial, i.e., only some part of the graphical representation is changed while preserving others. (Of course, there are situations where a small modification can drastically change the entire program behavior. A change in an algorithm is an example. We do not consider such situations.)

A problem is that there is no simple way to identify changed and unchanged elements in object graphs obtained from the executions of the two versions of the program. (There are possible approaches, for example to compute a maximal graph matching, which is computationally expensive.)

Here, we propose a novel technique, called *the calling-context sensitive object identification*, to layout object graphs with preserving mental map. The technique is based the following ideas.

- We give a unique ID to each program location (precisely, we only maintain IDs for `new` expressions and method call expressions.) We preserve the ID even if the programmer edits the program text.
- We give a context-sensitive ID of an execution of an expression. A context-sensitive ID of an execution of an expression is a triple of a program location ID, execution counts of (syntactically) enclosing loops, and the context-sensitive ID of the execution of a method call expression that invoked the enclosing method of the current execution.
- When there is an execution of a `new` expression, we use the context-sensitive ID of the execution as the ID of the created object.
- When we draw an object graph obtained from an execution of a modified program, we layout it so that an object will be placed at the same position of an object with the same ID in the execution of the previous program.

Intuitively, we consider two objects are 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 counts.

[8] http://visjs.org/

6 RELATED WORK

Among the programming environments with data visualization, Python Tutor [6] has a 'live' feature. When a program is edited, it automatically re-executes the program and updates the visual representations. However, there are several limitations. (1) There is no mechanism to synchronize the cursor position and the program execution point to visualize. Hence the developer needs to specify a program point only by using forward and backward buttons. (2) Visual representation of data structures is not stable when a program is changed. (3) A point of program execution to visualize is also not stable. As far as the authors observed, when the system is showing at the N-th step a program, it will show, when the program is changed, at N-th step of the modified program. Therefore, a change to the code that has been executed until the N-th step can lead the system to an unexpected point of execution.

The Morphic environment in Self [15] lively integrates the code editor and the object inspector, which displays objects as a node-link diagram. The selection of visualized objects is manual. It always displays the *current* state of objects; i.e., it is not possible to show the paste state in an execution without resorting to a breakpoint debugger.

The calling-context sensitive identification technique (Section 5.2) can be used for problems that need to align executions of two versions of a program. For example, example-centric programming [4] uses a similar technique to identify a context of a test case that is created for an older version of a program.

Zimmermann and Zeller propose to display a visual object graph in a debugger [17]. They also propose to highlight difference between two object graphs. Their approach calculates a *correspondence graph* to determine the same objects.

Back-in-time debuggers [12] can show a state of a program at an arbitrary time in the execution. Similar to Kanon, they record program execution through program instrumentation. However, research in debuggers mainly focuses on recording more detailed information than mere object graphs. It is not clear if those techniques can be directly applied to live programming environments, where programs are frequently modified.

Kanon assumes that the programmer writes a program in a test-driven development [2] style. Such a style can be commonly found in live programming [9, 16] as well as in example-centric programming [4].

7 CONCLUSION

We propose Kanon, a live data structure programming environment. The notable features are the mechanisms that help connecting the visual representation and the program code. The snapshot view mode shows the object graph when the program execution is reached at the cursor position, which should be relevant to the programmer's mental state. The summarized view mode shows the effect of an expression under the cursor throughout the execution. This helps the programmer to check whether the current expression behaves as expected. A probe on a variable helps connecting a variable in the program to a graphical element. The jump-to-construction mechanism helps connecting a graphical element to an element in a program. Finally, we proposed *the calling-context sensitive object identification technique* to preserve mental map of representation between the graphs generated by modified programs.

Future work includes development of a better object graph layout algorithm that is closer to the programmer's expectations, improvement of feedback performance, and a selective visualization mechanism for larger object graphs.

ACKNOWLEDGEMENTS

The authors would like to thank Jun Kato for his valuable comments. This work was supported by JSPS KAKENHI Grant number 26330078.

REFERENCES

[1] Samuel Aaron 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)*. 35–46.

[2] Kent Beck. 2003. *Test-Driven Development: by Example*. Addison-Wesley Professional.

[3] Apple Computer. Swift Playgrounds. http://www.apple.com/swift/playgrounds/. Accessed Ferburary 2017.

[4] Jonathan Edwards. 2004. Example Centric Programming. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, 84–91.

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

[6] Philip J Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. 579–584.

[7] Christopher Michael Hancock. 2003. *Real-time Programming and the Big Ideas of Computational Literacy*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

[8] T. Dean Hendrix, James H. Cross, II, and Larry A. Barowski. 2004. An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE. *SIGCSE Bull.* 36, 1 (March 2004), 387–391.

[9] Tomoki Imai, 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* (2015-10-25), 11–12.

[10] Jun Kato, 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)*. 189–196.

[11] Khan Academy. Intro to JS: Drawing & Animation. https://www.khanacademy.org/computing/computer-programming/programming. Accessed Ferburary 2017.

[12] Bil Lewis. 2003. Debugging Backwards in Time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*. arXiv:cs.SE/0310016 http://arxiv.org/abs/cs.SE/0310016

[13] Henry Lieberman 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)*. ACM Press/Addison-Wesley Publishing Co., 480–486.

[14] Sean McDirmid. 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!)*. 623–638.

[15] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87) (ACM SIGPLAN Notices)*, Vol. 22(12). 227–242.

[16] Bret Victor. 2012. Inventing on Principle. Keynote Talk at the Canadian University Software Engineering Conference (CUSEC). (Jan. 2012).

[17] Thomas Zimmermann and Andreas Zeller. 2002. Visualizing Memory Graphs. In *Software Visualization*. Springer, 191–204.