On Defining Recursive Functions in Live Data Structure Programming

Akio Oka[1], Hidehiko Masuhara[2], Tomoyuki Aotani[3]

Tokyo Institute of Technology
[1] a.oka@prg.is.titech.ac.jp
[2] masuhara@acm.org
[3] aotani@is.titech.ac.jp

Abstract

Kanon is a live programming environment that automatically visualizes data structures created by the program being edited. 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. This paper proposes an extended feature of Kanon that lets the programmer manually build an expected structure. The expected structure not only makes the visualization useful again, but also serves as a test case. This paper also discusses the usefulness of the extended feature through case studies.

1 Introduction

Live programming [8, 21] 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.

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 [21], for synthesizing music [2], for animating game characters [16], and for teaching algorithms [14].

Data structure programs 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.

We proposed a live programming environment, called Kanon, specialized for data structure programming [19, 17, 20, 18], and carried out a user experiment to collect programmers' opinions. We observed that, though most of the participants had positive impressions, Kanon still has room for improvement.

One of the findings from the experiment is that the visualization is mostly useless for defining recursive functions.

In the following sections, we discuss the background of this research necessary to explain the proposing features (Section 2). We then describe the design of our proposal features (Section 3) and verify usefulness of the proposed features (Section 4). We explain an implementation of the features (Section 5). Finally, we discuss related work (Section 6) and conclude the paper (Section 7).

2 Background: A Live Data Structure Programming Environment

We proposed Kanon, a live programming environment for data structure programming [19, 17, 20, 18]. Here, we introduce notable features of Kanon that are relevant to define recursive functions.

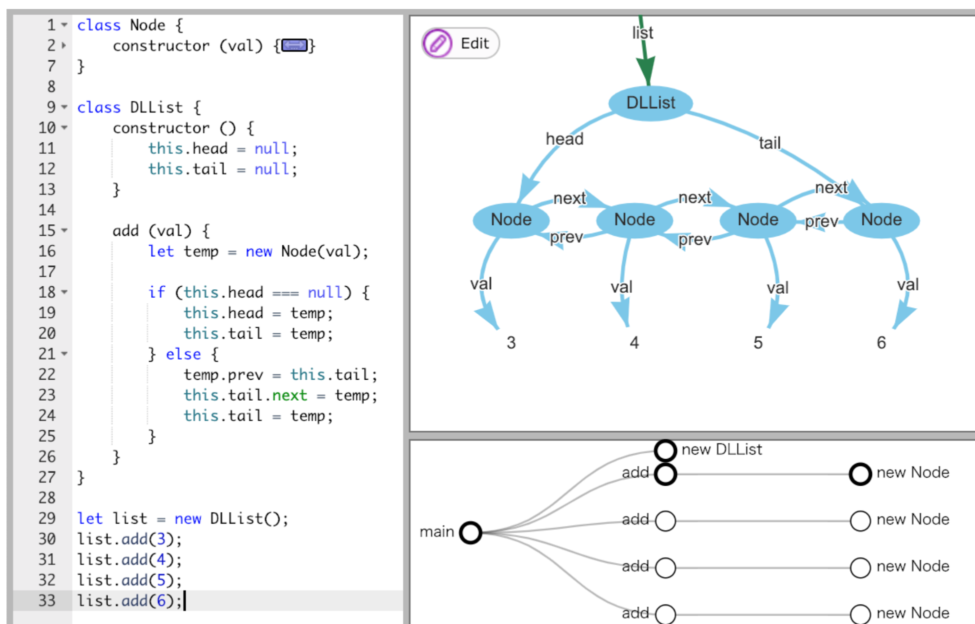

Figure 1. A screenshot of Kanon.

2.1 Design Overview and Assumptions

Figure 1 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 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.

2.2 Visualization of Changes and View Mode

2.2.1 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

[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.

such changes as a series of values [16, 11] or as a line chart [3]. For programs that produce visual images (i.e., drawing programs), there have been attempts to use a stroboscopic visualization [7] or a timeline visualization [13].

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.

2.2.2 Snapshot View Mode

Kanon provides two ways of showing changes in a program run. Here we will introduce one of the ways, called the *snapshot view mode*, which animates the graphical representation using cursor movement in the text editor.

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 is an example of a snapshot view for the program text in Listing 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, 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.

Listing 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);
```

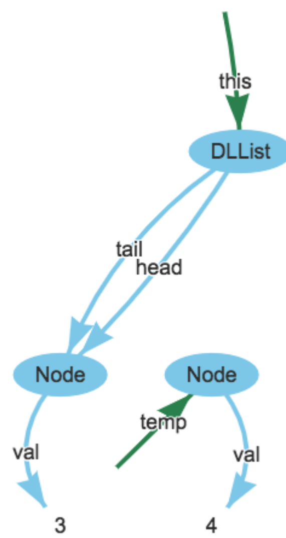

Figure 2. A snapshot view in the context of `l.add(4)`.

2.3 Control Flow Graph

Kanon provides a call tree pane that displays which function calls which function as shown in the lower right in Figure 1. 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

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

the snapshot view mode. In the case of Figure 1, 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 1.

3 Recursive Function Definition in Live Data Structure Programming

In this section, we propose features which solve one of the problems of existing Kanon. First, we will describe the problems when defining recursive function with existing Kanon. Second, we will propose a mechanism to solve the problems.

3.1 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.

3.1.1 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 2 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.

Listing 2. 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 }
```

3.1.2 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 3, partially wrote `reverse` as shown in Listing 2, and then located the 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 3) 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 3. 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();
```

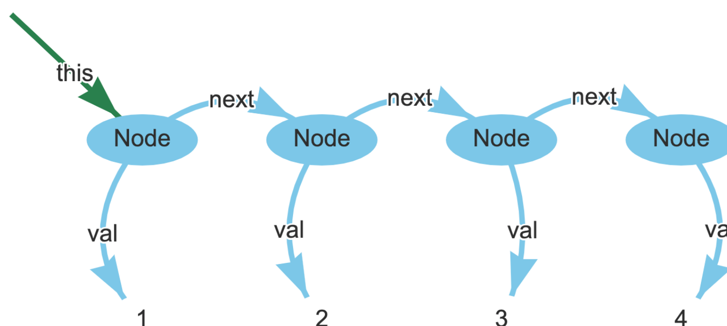

Figure 3. A node-link diagram displayed by the existing Kanon just after writing the recursive function call in Listing 2.

3.1.3 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.

3.2 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.

3.2.1 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.

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 2 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 3.

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 shows a screenshot of Kanon with a window to edit the graph to specify the expected structure[3]. 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[4] on the upper right corner of the window registers the diagram as an expected structure[5].

3.2.2 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.

3.2.2.1 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 5).

Additionally, the result is also displayed on the visualization of the call tree (like Figure 6.) 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.

---

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

[4]Clicking the *actual graph* button in Figure 4 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.

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

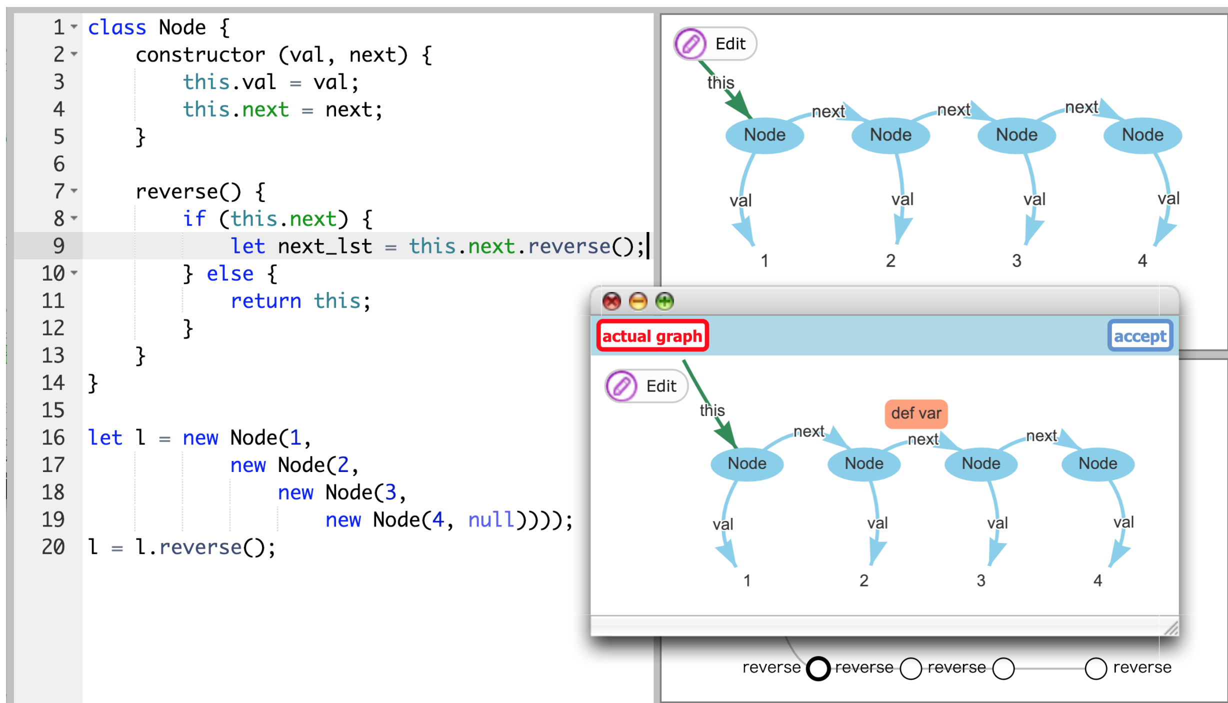

Figure 4. A pop-up window for building an expected structure. The red rectangle *def var* is a button to introduce a variable reference (like the green arrow with `this`) into the diagram.

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

Figure 5. A function call which an expected graph is inserted and its test failed.

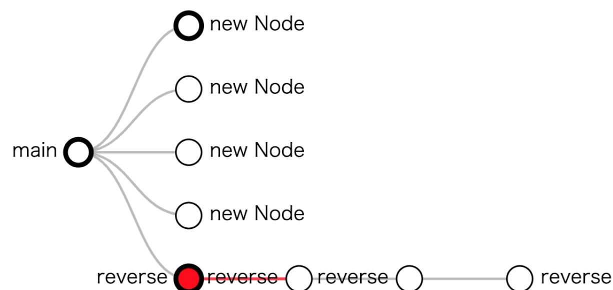

Figure 6. A call tree with test highlight.

3.2.3 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.

We assume that the programmer built the expected structure as shown in Figure 7. 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.

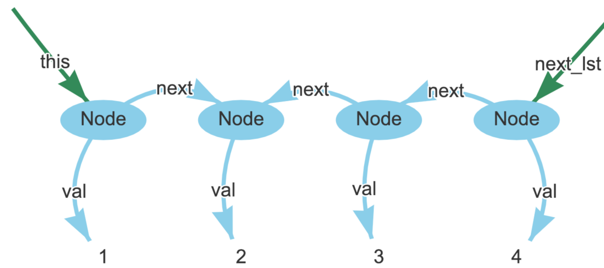

Figure 7. An example of a graph after overriding object structure.

4 Case Studies

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. At the time, we expected the structure after executing the recursive call as shown in Figure 7, and built and registered the structure. After that, Kanon shows the graph as shown in Figure 7 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.

```
1  class Node {
2      constructor (val, next) {↔}
6
7      reverse() {
8          if (this.next) {
9              let next_lst = this.next.reverse();
10             this.next.next = this;
11             this.next = null;
12             return next_lst;
13         } else {
14             return this;
15         }
16     }
17 }
18
19 let l = new Node(1,
20         new Node(2,
21             new Node(3,
22                 new Node(4, null))));
23 l = l.reverse();
```


Figure 8. A screenshot of Kanon after the programmer has completed defining the recursive function `reverse`.

Figure 8 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 8 become green, we immediately understood that the definition is correct (at least with respect to the test case).

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 9 shows the screenshot at this point.

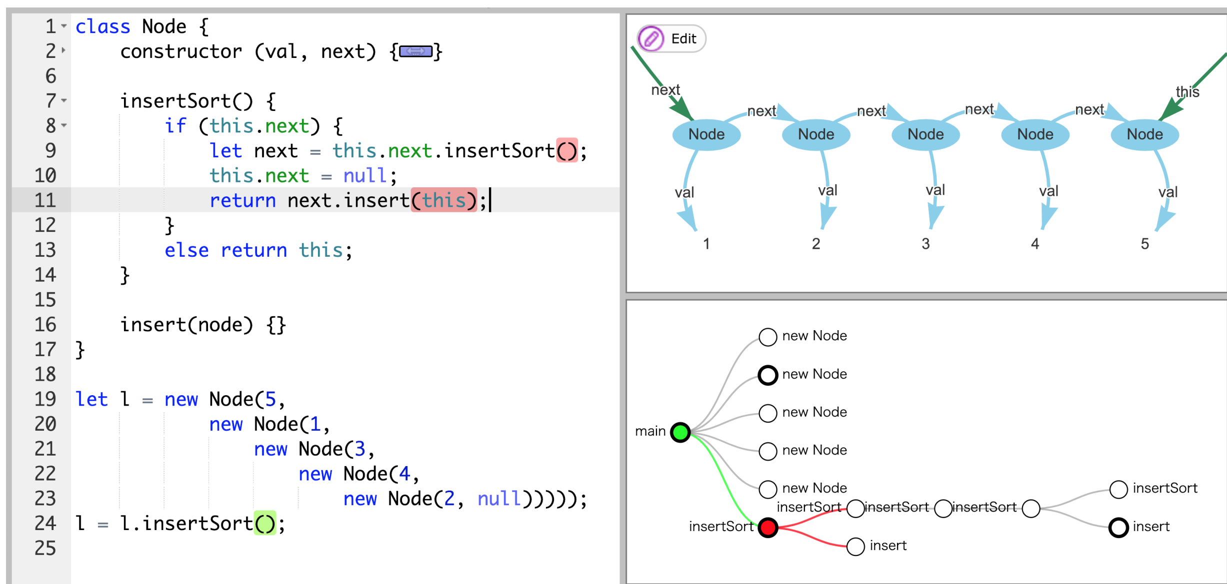

Figure 9. A screenshot after writing the body of `insertSort`.

Interestingly, the call tree in Figure 9 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 10. It was easy to add them since we only needed to modify part of the list.

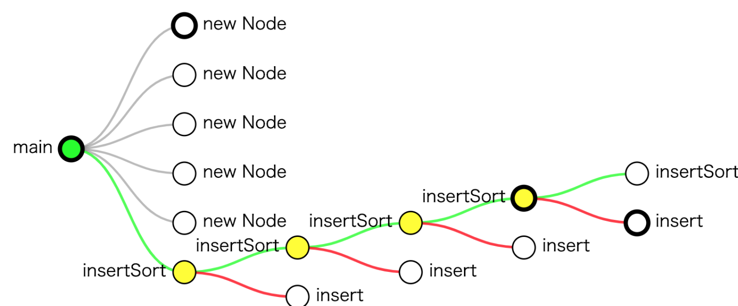

Figure 10. A call tree after the programmer has inserted expected structures to the call, written at line 9 and line 11 on Figure 9, on all contexts.

Finally, we wrote the body of `insert` by selecting one of four registered test cases in turn. The final screenshot is Figure 11. 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.

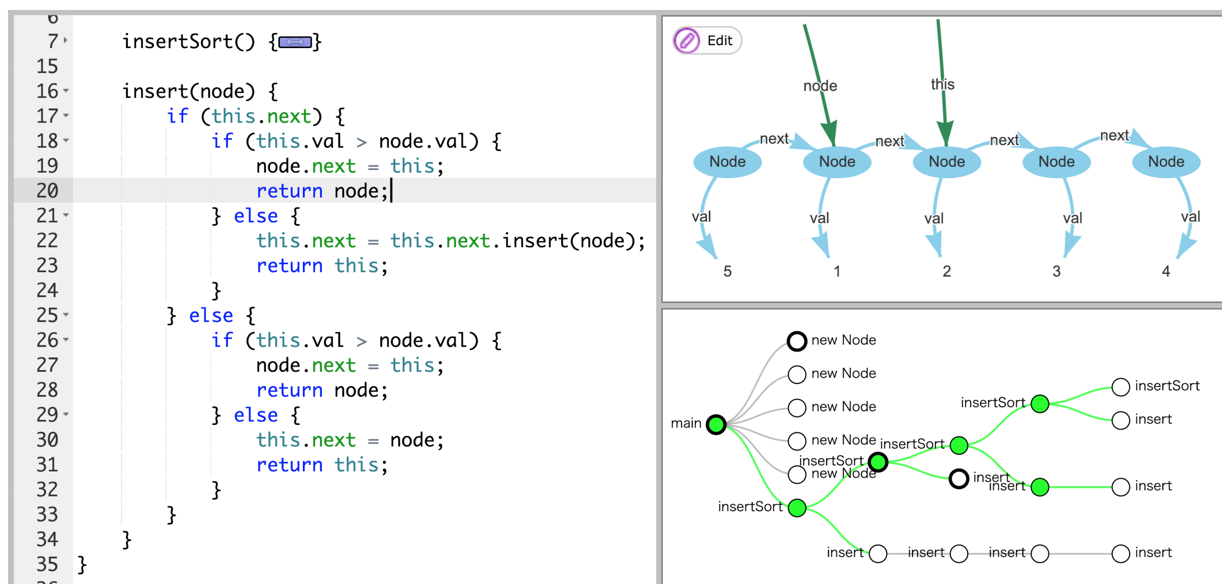

Figure 11. A screenshot when the `insert` method succeeded in all the tests.

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.

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()`[6]. 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 12 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(`$\langle\langle z\rangle\rangle$`)`.

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

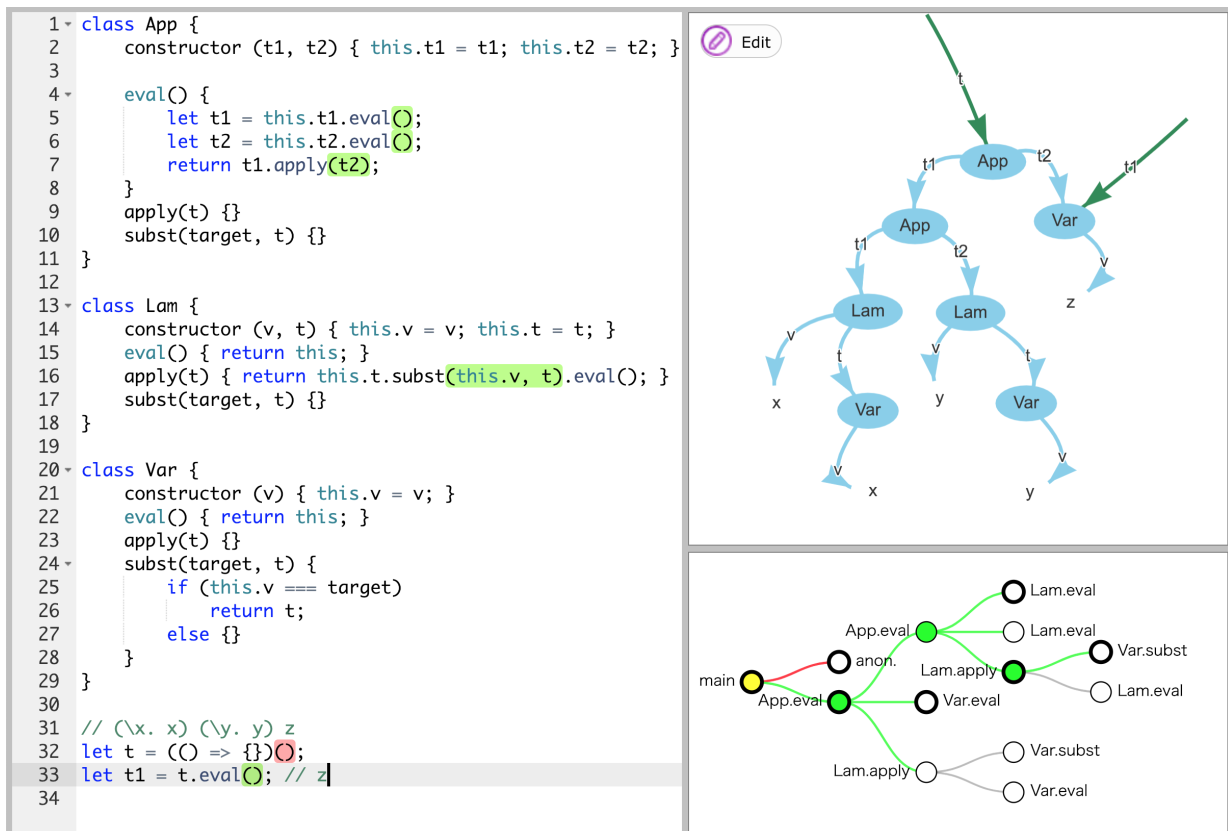

Figure 12. A screenshot after we use just one test case to define the methods of `eval`, `apply`, and `subst`.

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.

5 Implementation

We implemented a prototype of Kanon for JavaScript running on web browsers. It is available online[7]. Below, we first overview the implementation and then describe how to realize the automated

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

testing and overriding.

5.1 Overview

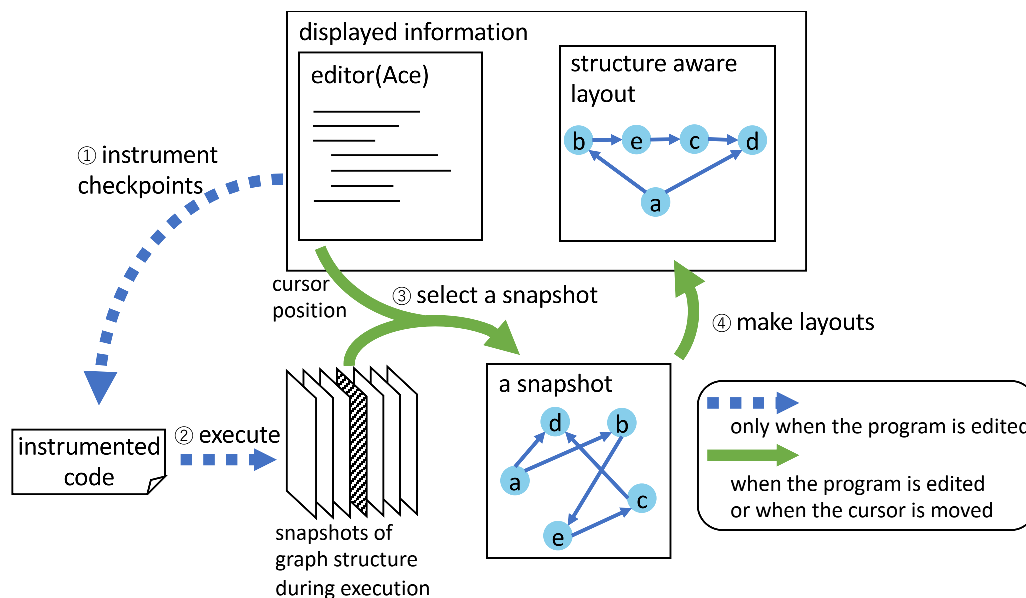

Figure 13. Overview of the implementation.

Figure 13 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 [12] for editing program text. When the programmer edits a piece of text, it notifies the visualization engine.

① The visualization engine uses Esprima [10] 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.
- 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 closest checkpoint before the cursor position in the *snapshot view mode*. 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 closest checkpoint before the current cursor position within the calculated timestamp range.

④ The engine computes the 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. At that time, the newly created nodes will be placed aesthetically-pleasing positions by using the vis.js visualization library [5] for calculating the layout.

5.2 Context Management

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[8].

To preserve the mental map when a program is modified, Kanon equips a technique, called *calling-context sensitive identification*, for preserving mental map of object graphs. The technique gives a calling-context based identifier, called *context-sensitive ID*, to each function call, and records each object graph by associating 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.

The technique 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.

5.3 Automated Testing and Overriding

5.3.1 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 [1]. In the window, the manipulation mechanism utilizes the feature provided in vis.js library [5], 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

[8]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 [4, 15] because the human who saw a graph would need a lot of time to grasp the structure of the graph.

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.

5.3.2 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 5.1, Kanon converts function calls as shown in Listing 4. in Listing 4, 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 5.3.3). 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. 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`.

5.3.3 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.)

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, 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.

5.3.4 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.

5.3.4.1 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, and the programmer has put the expected structure as shown in Figure 7 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. 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 7, the programmer expects a structure where the new element has been inserted to Figure 7. 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.

5.3.4.2 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[9]. The programmer needs to confirm the warned structure to remove the warning.

6 Related Work

EG is an Eclipse plug-in for example-centric programming [6], 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 [6] does not explicitly discuss the need for the overriding feature at defining recursive functions, even though its running example is a recursive factorial function.

Vital is an interactive graphical environment for Haskell [9], 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.

7 Conclusion

We proposed a set of features for defining recursive functions in a live data-structure programming environment Kanon. With the features, the programmer can enjoy live programming experience

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

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`.

Kanon still has more future improvements. We should design features that the programmer can define methods or functions that works on arbitrary input data. Additionally, Kanon should be able to provide users with alternative ways of testing such as partially testing that the user can partially specify expected parts of the structure.

References

[1] Prototype Window Class. `https://github.com/sgruhier/prototype_window`. Accessed January 2019.

[2] Samuel Aaron and Alan F. Blackwell. 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, pages 35–46, New York, NY, USA, 2013. ACM.

[3] Apple Computer. Swift playgrounds. `http://www.apple.com/swift/playgrounds/`, 2016. Accessed February 2017.

[4] D. Archambault and H. C. Purchase. The mental map and memorability in dynamic graphs. In *2012 IEEE Pacific Visualization Symposium*, pages 89–96, Feb 2012.

[5] Almende B.V. vis.js - a dynamic, browser based visualization library, 2018.

[6] Jonathan Edwards. Example centric programming. In Doug Schmidt, editor, *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 84–91, New York, NY, USA, 2004. ACM.

[7] Chris Granger. Light Table. `http://www.chris-granger.com/lighttable/`, 2012. Accessed February 2017.

[8] Christopher Michael Hancock. *Real-time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2003.

[9] Keith Hanna. Interactive Visual Functional Programming. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 145–156, New York, NY, USA, 2002. ACM.

[10] Ariya Hidayat. Esprima, 2018.

[11] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. Making live programming practical by bridging the gap between trial-and-error development and unit testing. In Jonathan Aldrich, editor, *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 11–12. ACM, ACM, October 2015.

[12] Fabian Jakobs. Ace - the high performance code editor for the web, 2018.

[13] Jun Kato, Sean McDirmid, and Xiang Cao. 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)*, pages 189–196. ACM, 2012.

[14] Khan Academy. Intro to JS: Drawing & animation. `https://www.khanacademy.org/computing/computer-programming/programming`, 2018. Accessed February 2017.

[15] Yi-Yi Lee, Chun-Cheng Lin, and Hsu-Chun Yen. Mental map preserving graph drawing using simulated annealing. In *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation - Volume 60*, APVis '06, pages 179–188, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[16] Sean McDirmid. 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!)*, pages 623–638. ACM, 2007.

[17] Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. The Visualization of Data Structures and Interactive Features for Live Programming. In *The 113th IPSJ Workshop on Programming*, March 2017. Japanese.

[18] Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. 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, pages 72–87, New York, NY, USA, 2018. ACM.

[19] Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. Kanon: Data Structure Programming using Live Programming Environment. In *Proceedings of the 10th JSSST Workshop on Programming and Programming Languages*, PPL 2017, March 2017. Japanese.

[20] Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. Live Data Structure Programming. In *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, pages 26:1–26:7, New York, NY, USA, 2017. ACM.

[21] Bret Victor. Inventing on principle. Keynote Talk at the Canadian University Software Engineering Conference (CUSEC), January 2012.