Journal of Information Processing Vol.30 451–463 (June 2022)

[DOI: 10.2197/ipsjjip.30.451]

Regular Paper

An Approach to Collecting Object Graphs for Data-structure Live Programming Based on a Language Implementation Framework

Shusuke Takahashi[1,a)] Yusuke Izawa[1,b)] Hidehiko Masuhara[1,c)] Youyou Cong[1,d)]

Received: August 31, 2021, Accepted: January 11, 2022

Abstract: Data-structure live programming environments execute programs, collect object graphs (objects and their mutual references) created and modified during the execution, and visualize the graphs as a node-link diagram. Existing implementations collect object graphs by instrumenting checkpoints, at which the system traverses reachable objects, at every necessary point in the program. Since the cost of each checkpoint is proportional to the number of existing objects, the overhead of running checkpoints can be huge. This paper proposes (1) a technique to collect object graphs by recording object creation and modification events into an efficient data structure, and (2) an implementation design for the object graph collection mechanism by extending a language implemented on top of a language implementation framework. As a result, the overhead of object graph collection is almost proportional to the number of object creation/modification operations in total. We implemented the proposed mechanism for the Kanon data-structure live programming environment by extending GraalJS, a JavaScript implementation on the Graal/Truffle language implementation framework. We compared our new implementation against the original Kanon, which is based on checkpointing, and confirmed that our implementation improves program execution (and data collection) speed, and has sufficiently small overheads to reconstruct object graphs.

Keywords: live programming environment, meta-language implementation framework, Graal, Truffle, object graph

1. Introduction

A live programming environment is a piece of software that reruns a program as soon as the user edits it, and displays the content of the execution. This feature allows the user to see immediately the effect of a minute edit on the program [1]. Examples of existing live programming environments include Scratch [2]; Dejavu [3], which supports interactive development using camera input; programming environments for music [4]; and Kanon [5], which is specialized in visualizing data structures.

Kanon (**Fig. 1**) is a live programming environment for visualizing data structures as a graph automatically and immediately while editing the program. When the user writes a program that handles data structures with complex reference relations, Kanon helps the user by correcting instructions as the user writes them to update each element of the data structures. We assume that the data to be visualized are usually small when creating a program. However, in practical software development, there is the possibility of hundreds of objects being handled.

To perform visualization, a data structure live programming environment such as Kanon must collect the objects that exist at each point of the program execution and assess the relations that references bear. There are two methods for performing those features: one is the *snapshot technique*, which makes a copy of the reference relations between the objects that exist at each point in time, and the other is the *operation history technique*, which records instantiations or modifications of an object whenever they are performed by the program, and it restores the reference relations between the objects at a certain point in time from the record when the relations are needed.

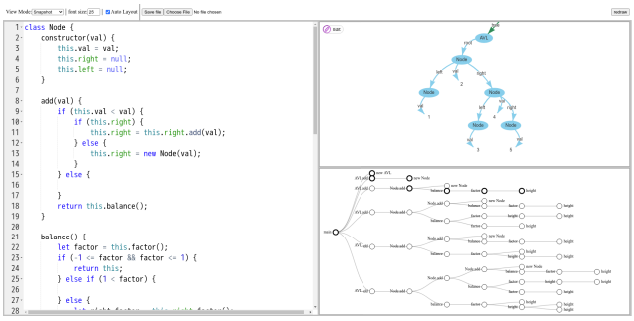

Fig. 1 Screenshot of Kanon. On the left is the editor. On the upper-right is the node-link diagram based on an object graph. On the bottom-right is a tree representing function calls and iterative structures at runtime.

Kanon implements the snapshot technique with source code conversion. During conversion, it inserts checkpoints before and after each statement in the given source code. After that, it executes the converted program. When the checkpoint is executed, Kanon records all the reachable objects in the scope of the checkpoint using the objects that the variables refer to and their reference relations. This method can be implemented without any

[1] Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[a)] takahashi@prg.is.titech.ac.jp
[b)] izawa@prg.is.titech.ac.jp
[c)] masuhara@acm.org
[d)] cong@c.titech.ac.jp

specialized language processing system. On the other hand, the time taken by the checkpointing process is proportional to the size of the reachable object graph. As mentioned above, we assume that Kanon visualizes hundreds of objects, so this overhead should not be ignored.

In this paper, we propose an efficient implementation of the object operation history technique. In a naïve technique, the size of the information recorded at the time of object instantiations and modifications is of constant order. However, it takes time proportional to the number of creations and modifications to recover the reference relations of objects from the records. We propose a method to record the instantiations and modifications of objects as intermediate representations (IR) and recover object reference relations efficiently at arbitrary points in time from these IR.

To collect the history of operating objects, it is necessary either to perform program conversion or create an original language processing system. In this study, we propose using a meta-language implementation framework to create an original language processing system.

The proposed method, which uses the meta-language implementation framework, has the following advantages. First, the program conversion method has the problem that it is impossible to record object operations occured in programs that do not have source code, such as libraries. In contrast, the proposed method can solve this problem. Secondly, it is possible to implement an original language processor with only minor changes by modifying an existing language processor in a language implementation framework. Finally, most of the mechanisms for collecting operation history, which we plan to propose in the future, can be implemented in a common way for multiple languages. This is because object operations are often defined as a method shared among multiple languages in language implementation frameworks where multiple languages are available.

In this paper, we experimentally implement an object graph generation system based on the proposed method using Graal/Truffle [6], [7], a language implementation framework, and GraalJS, a JavaScript interpreter implemented with the framework. The application with multiple languages is beyond the scope of this paper because we implement the system on the assumption of using GraalJS.

This paper is structured as follows. In Section 2, we describe Kanon, a live programming environment, and Graal/Truffle, a language implementation framework. In Section 3, we present our method of collecting object operation histories using a language implementation framework, a transformation from operation histories into IR, as well as a concrete implementation for generating object graphs using the IR. In addition, we discuss in Section 4 what is unique to our experimental implementation using Graal/Truffle and GraalJS. In Section 5, we compare the performance of object graph collection with that of the original Kanon. We list related work in Section 6, and Section 7 concludes this paper and discusses future prospects.

2. Background

As previously mentioned, this paper presents an improved version of the object graph collection mechanism in Kanon. This approach is realized by extending the GraalJS interpreter, which is implemented using the language implementation framework Graal/Truffle. This section provides an overview of the implementation of Kanon, introduces the fundamental mechanism, and briefly describes the architecture of Graal/Truffle and GraalJS.

2.1 Kanon

Kanon is a live programming environment specialized in visualizing data structures. It re-runs a program in the editor (the left side of Fig. 1) each time the user edits it, and displays the object graph (upper-right of Fig. 1) built when the program is executed up to the cursor position in the editor. Kanon allows the user to develop a program based on the displayed diagrams and ensure it is implemented as intended.

In addition, Kanon not only shows the reference relations of objects but also allows the user to zoom in and out of the diagram and change the position of the nodes. Furthermore, when Kanon redraws the diagram due to changes in the source code, it tries to maintain the last drawn layout.

2.1.1 Execution Process of Kanon

Figure 1 shows the execution flow of Kanon—from the input of the program in the editor to the drawing of the node-link diagram.

(1) First, Kanon converts the given program to catch run-time information. Specifically, it parses the entire program and inserts checkpoints before and after every statement in the program.

(2) It then executes the converted program with the `eval` function. When the inserted checkpoint is executed, it collects the information explained in Section 2.1.2 by the following procedure.

- (i) **Reference relations of objects**: Kanon records all of the reference relations of objects. Specifically, it obtains all local and global variables, and transitively records all objects that exist at that time.
- (ii) **Location in the source code corresponding to an object instantiation and modificiation**: Kanon records the location of the checkpoint in the source code.
- (iii) **Call stack when the object is generated**: First, Kanon obtains the call stack when the checkpoint is executed. Then, it links the newly created object to the call stack by taking the difference with the object references obtained in (i).

(3) Kanon gets the cursor position and selects the information obtained at the nearest checkpoint.

(4) Kanon generates a node-link diagram based on the selected information and displays it.

When the user moves the cursor in the Kanon editor, steps (3) and (4) are performed again (corresponding to the dashed line in **Fig. 2**). In this case, Kanon reuses the information obtained from the latest edited program. When the user edits the program, Kanon executes steps (1) and (2) again (corresponding to the solid line in Fig. 2), followed by steps (3) and (4).

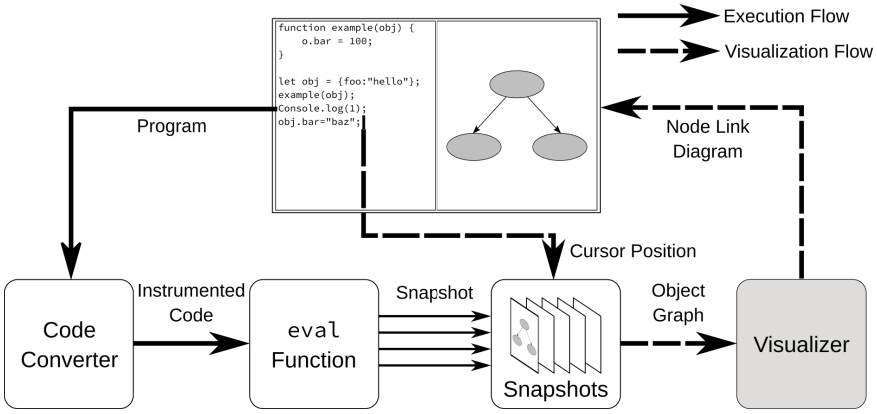

Fig. 2 Execution flow of Kanon.

2.1.2 Information Collected by Kanon during Program Execution

To generate an object graph, Kanon records run-time information during program execution. It requires the following three kinds of information for object graph generation, as explained in Section 2.1.1: (i) reference relations of objects, (ii) location in the source code corresponding to object instantiation and modification and (iii) call stack when the object is created. The following paragraphs describe each of these three kinds of information.

(i) Reference Relations of Objects

Kanon defines *the reference relation of each object* as the relation by which objects refer to each other. This object relationship of every object at some point can generate the following three kinds of value during program execution:

- value of local variables,
- value of global variables,
- value of objects and arrays.

We take the example shown in **Fig. 3** to explain how referencing relations are generated. Here, we denote the object initialized by the right side of the second line as `A`. In this case, the value of the local variable `object` is `A`, the value of the global variable `x` is the number 1, and object `A` has the number 1 in the field `f`.

(ii) Location in the Source Code Corresponding to Object Instantiation and Modificiation

Kanon records the expressions or statements in the source code that perform object instantiation and manipulation. When the user places a cursor at a specific source code position, Kanon visualizes the object graph up to that position.

(iii) Call Stack when the Object is Generated

Kanon has a mental-map preserving [5] mechanism. A mental map is a representation constructed in a programmer's brain when they see visual information such as graphs and pictures. Mental map preservation [8] is a function to keep the original form as much as possible so as not to redraw the produced visual images from the beginning. By minimizing the graph layout changes, Kanon makes it easier for a programmer to grasp the differences before and after program modifications.

Kanon stores the following information to realize mental map preservation: (1) node positions in a last displayed graph and (2) the nodes that should be associated. The former is beyond the scope of this paper, but we describe the latter here. To map

```
1 function example() {
2     let object = {f: 1};
3 }
4
5 let x = 1;
6 example();
```

Fig. 3 Example of JavaScript program with variables and objects.

```
1 function example() {
2     return {f: 1};   ▲
3 }
4                      // Function call ID
5 let x = example();   // example1
6 let y = example();   // example2
```

Fig. 4 Example of JavaScript program showing the difference of call stacks when objects are generated.

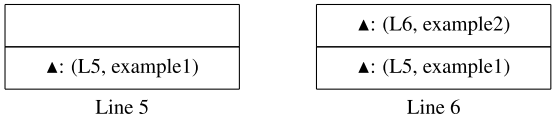

Fig. 5 Call stacks for each function call in Fig. 4.

objects, Kanon not only records the location in the source code where the object is generated but also assigns a unique ID to each function call. By recording both the generated object and the pair of a source code position and a function call ID (which is called the context of function call here) in the **call stack**, it is possible to determine which function call generates an object. This technique also enables objects to be drawn separately even if the same function is called in a different context. Furthermore, after program modification happens, it allows the previous node's position to be maintained by resuming the function call history based on a recorded call stack, rather than rebuilding it from the beginning.

Figure 4 describes what the call stack is[*1]. The values of global variables `x` and `y` are both objects ▲ created when the statement in Line 2 is executed. When `x` is instantiated, the call stack records the information that function `example` is called at Line 5 (▲: `(L5, example1)`). Furthermore, when `y` is instantiated, the call stack records the information that function `example` is called at Line 6 (▲: `(L2, example2)`). When Line 6 is executed, the call stack ends up as in the right-hand side of **Fig. 5**. Through

[*1] As this paper focuses on recording the context of a function call as a call stack, we omit the description of the mapping between the call stack before and after editing.

the operation of a call stack, Kanon can distinguish the object generated by each function call with the call stack.

2.2 Graal/Truffle

A language implementation framework [6], [9] is a toolchain that generates virtual machines with an efficient memory model, a high-performance garbage collector, and a fast JIT compiler. It is used to implement PyPy [9] and TruffleRuby [10], both of which show higher performance than CPython and CRuby, respectively.

When language developers use a language implementation framework, they need to define the interpreter of the language they are going to implement (guest language) in the framework language (host language). The framework takes this interpreter as input and generates a high-performance virtual machine with a dedicated garbage collector, JIT compiler, etc.

In particular, Graal/Truffle partially optimizes a given guest language interpreter, using a technique called *self-optimizing interpreter*, to run guest language programs faster on a virtual machine called GraalVM.

2.2.1 GraalJS

GraalJS is a JavaScript interpreter, written in Graal/Truffle, that runs on GraalVM. This section describes how GraalJS is implemented with Graal/Truffle.

In general, Graal/Truffle requires language developers to write abstract syntax tree (AST) interpreters. The same goes for GraalJS. Each AST node in GraalJS is defined as a class inheriting the `Node` class provided by Graal/Truffle. The `Node` class has an abstract method `execute` that is called when executing a node, and language implementors should implement the method `execute` in each node class.

All nodes in GraalJS are defined by inheriting the class `JavaScriptNode`, which inherits the `Node` class. For example, a binary operator + for integer values in GraalJS is implemented as shown in **Fig. 6**. This node has members representing the left-hand side and the right-hand side. The method `execute` first evaluates each sides and returns the sum. Here, the argument `frame` in the `execute` method represents the stack frame when this node is executed.

The class `Node` also has a method `getSourceSection`, which returns the description corresponding to the node in the source code.

2.2.2 Objects in Graal/Truffle

In Graal/Truffle, objects in a guest language are represented using the following three classes: `DymanicObject`, `Shape`, and `Property` [11], as shown in **Fig. 7**.

A subclass of `DynamicObject`, which is an abstract class provided by Graal/Truffle, holds all values in an object. For implementation, a developer usually implements the subclass in the following ways: (1) holding the value of an object directly as a member variable, or (2) referencing to some data structure such as an array including values.

The class `Property` represents a pair of field identifiers and their corresponding values in `DynamicObject`. It is possible to update a value by passing an instance of `DynamicObject` to operate with the class `Property`.

The class `Shape` has as many instances of `Property` as fields in the object as its member variables.

```
1  class IntPlusNode extends JavaScriptNode {
2      JavaScriptNode lhs;
3      JavaScriptNode rhs;
4
5      int execute(Frame frame) {
6          int lhsResult = lhs.execute(frame);
7          int rhsResult = rhs.execute(frame);
8          return lhsResult + rhsResult;
9      }
10 }
```

Fig. 6 Example class of a subclass of `JavaScriptNode`.

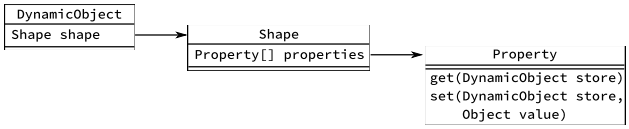

Fig. 7 Class diagram showing the replationship between `DynamicObject`, `Shape`, and `Property`.

2.2.3 Implementing Tools with Instrumentation API

Instrumentation API in Graal/Truffle is a set of features for developing tools running on GraalVM. Thanks to it, when an interpreter executes a program, it enables getting the node that is currently running and inserting another process, which is defined by a tool developer, before and after the execution of the node.

Among the things that a tool developer can do with Instrumentation API is to put another process between guest language's node executions. As an implementation in a tool, a tool developer specifies a node that the process targets and when to execute. Specifically, a tool developer defines a class that inherits the class `ExecutionEventNode` and a class inheriting the interface `ExecutionEventNodeFactory`. A child of `ExecutionEventNode` defines a process inserted between the executions of nodes and how to handle an exception. A class implementing `ExecutionEventNodeFactory` specifies an assignment of `ExecutionEventNode` to a node. A tool developer can use the following properties for specification: file name, source code position, and tag label representing node's characteristics. In particular, a tag label indicates the node's property, whether it is a statement, an expression, or a function call.

For example, we consider a tool that produces a specified string before and after each node is executed. **Figure 8** shows an implementation example of the tool. The class `ExampleNode`, which is a subclass of `ExecutionEventNode` (Line 10 and later in Fig. 8), describes what to do before and after the execution of the node. The method `create` defined in the class `ExampleNodeFactory`, which inherits `ExecutionEventNodeFactory`, specifies what kind of process is performed on a node based on information that the instance of the class `EventContext`, which is passed as an argument, has. In this example, the method `create` returns an instance of `ExampleNode` when the context has the tag `Statement`; otherwise, it is `null`. When the program that is listed in the first line of **Fig. 9** is executed with the tool, we obtain the outputs shown after the third line of Fig. 9.

From the perspective of an AST interpreter definition, we need to follow the implementation discipline that the Instrumentation API provides. Specifically, the class representing an AST node should inherit not only the class `Node` but also the interface `InstrumentableNode` provided by the Instrumentation

```
class ExampleNodeFactory implements
     ExecutionEventNodeFactory {
    ExecutionEventNode create (EventContext context)
         {
        if (context.node.hasTag(Statement.class)) {
            return new ExampleNode();
        } else {
            return null;
        }
    }

    class ExampleNode extends ExecutionEventNode {
        void onEnter(Frame frame) {
            System.out.println("Entered");
        }

        void onReturnValue(Frame frame) {
            System.out.println("Exited");
        }
    }
}
```

Fig. 8 Example of tool implementation with Instrumentation API.

```
console.log("Executing");

// console output:
// Entered
// Executing
// Exited
```

Fig. 9 JavaScript program outputing "Executing" and the result obtained by executing the program with the tool shown in Fig. 8.

API. This `InstrumentableNode` interface requires the following two methods to be defined.

The first requested method is `createWrapper`, which returns a node converted so that it is possible to monitor its execution. The returned node must perform the process specified by the tool before and after executing the original node. This method takes an `ExecutionEventNode` as an argument. A tool developer has to implement the method `execute` of the nodes returned by the method `createWrapper` so that the method `execute` performs in the following order:

(1) execution of the method `onEnter` that is defined in the `ExecutionEventNode` instance passed as an argument,
(2) execution of the method `execute` of the original node,
(3) execution of the method `onReturnValue` that is defined in the `ExecutionEventNode` instance passed as an argument.

The second requested method is `hasTag`. This method is used in the `create` method in the `ExecutionEventNodeFactory` class. This `hasTag` method identifies what a node is. It takes a `Tag` instance as an argument and returns a boolean value indicating whether the node has a specific property or not.

Figure 10 shows an implementation using `InstrumentableNode`. The class `StatementNode` represents a node corresponding to a statement in a guest language. It implements `InstrumentableNode` and overrides `createWrapper` and `hasTag` methods. When the method `createWrapper`, which is listed after the second line, is called, an instance of an anonymous class that inherits the class `Node` is returned. The method `createWrapper` takes an instance of the class `ProbeNode`, which wraps an `ExecutionEventNode`. In the body of the `execute` method, the `onEnter` and `onReturnValue` methods are called before and after the original `execute` method. By this implementation, we can invoke the process defined by `ExecutionEventNode` before and after the execution of the original node. Also, the method `hasTag` ensures that the given tag equals `StatementNode`.

```
class StatementNode extends Node implements
     InstrumentableNode {
    Node createWrapper(ProbeNode pnode) {
        return new Node() {
            Object execute(Frame frame) {
                pnode.onEnter(frame);
                StatementNode.this
                    .execute(frame);
                pnode.onReturnValue(frame);
                ...
            }
        }
    }

    boolean hasTag(Class<? extends Tag> tag) {
        return tag == Statement.class;
    }

    Object execute(Frame frame) {...}
}
```

Fig. 10 Example of a class definition that implements `InstrumentableNode`.

3. Proposal of the Object Graph Collecting System

There are roughly two approaches to construct object graphs at each point of a program during execution: (1) storing the entire object graph generation histories (snapshot technique), or (2) storing the entire operation history, such as instantiation and modification of collected objects onto data structures like arrays, and then constructing graphs while drawing them (operation history technique).

The snapshot technique constructs object graphs at run-time. The time complexity of a run-time object graph generation is $O(N)$, where N is the total number of objects. The system must construct at least as many object graphs as the number of times the object is manipulated. Therefore, the time complexity of executing the program is at least $O(MN)$, where M is the total number of object operations. On the other hand, the system applies binary search on the snapshots generated during program execution to pick out an object graph. Thus the time complexity of selecting an object graph is $O(M)$.

The operation history technique only records operation history and constructs an object graph while drawing it. The time complexity is $O(M)$ because the system only has to collect object histories at run-time. Next, we consider the cost of constructing an object graph. The naïve approach replays the object history to an appropriate point for the object graph generation. This way requires at least $O(M)$ time.

In this paper, we propose a *time-series approach* that constructs object graphs using an intermediate representation (IR) that represents changes of the value in a field as a time series. For example, we consider a program such that the value in the field `x` is `1` at time 1. Its value changes to `2` at time 2. In this approach, the IR does not hold the record that `x = 1` and `x = 2` but the information whereby the value in the field `x` is `1` at time 1, and its value changes to `2` at time 2. In other words, our proposed IR contains a pair of a time and a value in each field. When constructing an object graph, this method replays the object history that is extracted from the IR. It can reduce the cost for object construction while

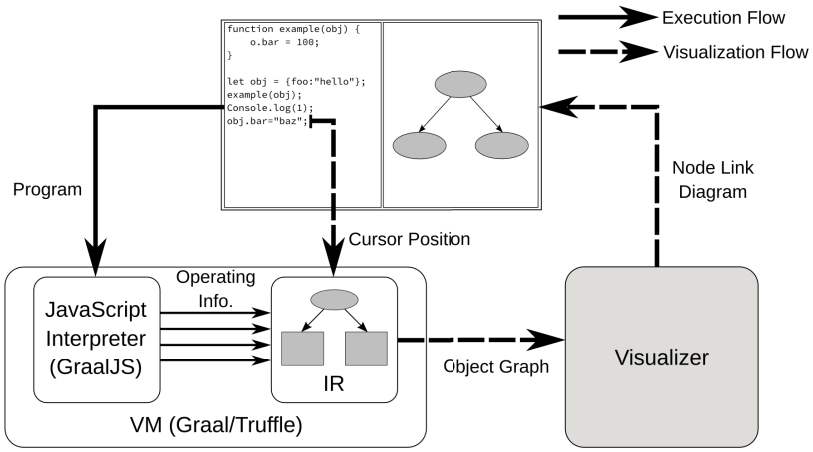

Fig. 11 Overview of the proposed system.

keeping the cost at execution equal to that of the naïve operation history technique.

Figure 11 shows an overview of our proposed object collection mechanism and the Kanon implementation using it. The solid and broken lines represent the flows of processes when a given program is executed, and an object graph is selected and visualized, respectively. Next, we describe how we implement our proposal.

First, we explain the definition of an *object operation history* in Section 3.1. This contribution corresponds to "Operating Info." in Fig. 11.

Second, we show the method of collecting object operations with a language implementation framework, which corresponds to "JavaScript Interpreter" in Fig. 11, in Section 3.2. We also present an implementation design to collect an object operation history in a language implementation framework. This step also corresponds to "JavaScript Interpreter" in Fig. 11. We also present an implementation design using a framework to collect an object operation history defined in the first step at run-time there.

Finally, we describe the IR based on an object operation history, and the method to build the object graph with it, in Section 3.3. This step corresponds to the IR in Fig. 11. We show that our proposal, which uses the IR, builds up an object graph faster than the naïve method using the operation history, especially in the case of handling a large number of object operations.

3.1 Recording Object Operations Used for Object Graph Collection

To generate object graphs, we need to gather the following information:

(1) object operation history, such as updating the value of a field contained in an object,
(2) source code section that is being executed when the object operation occurs,
(3) call stack when the object operation occurs.

3.1.1 Object Operation History

Our proposed approach records all changes, e.g., object initializations or updates, of each object's field value during execution. In particular, this approach stores the following information:

(1) object to be manipulated,
(2) identifier of the field whose value is being updated,

```
1 function f(x) {
2     x.fldA = 20;
3 }
4
5 let ob = { fldA: 100 };
6 ob.fldB = "hello";
7 f(obj);
```

Fig. 12 Sample code that manipulates an object.

(3) new value assigned to a field.

We consider the program shown in **Fig. 12**. The right-hand side of the assignment statement in Line 5 generates an object (we refer to this object as `ObjectA`). This assignment initializes the field at the same time and assigns the value `100` to `fldA`. Thus, the recorded information is (1) `ObjectA`, (2) `fldA`, and (3) `100`. An assignment to a field also occurs in Line 6. In the same way, the recorded information is (1) `ObjectA`, (2) `fldB`, and (3) `"Hello"`.

In addition to object operations, we also record variable assignments. To be specific, we keep tabs on the following three kinds of data:

(1) A variable area. We record it by seeing it as an object field. Taking an assignment to global variables as an example, we suppose that there is an object that has each global variable as a field. Then, the system records the operation as if it had occurred on the virtual object.
(2) A variable name.
(3) An assigned value.

We again consider the program shown in Fig. 12. In Line 5, `ObjectA` is assigned to the global variable `ob`. We assume that global variables are stored as fields in an object called `GlobalVariables`. In this case, the recorded information is (1) `GlobalVariables`, (2) `ob`, and (3) `ObjectA`.

3.1.2 Source Code Section

Live programming environments that visualize a data structure such as Kanon construct and visualize the object graph at the point specified by in the source code. To implement this feature, it is necessary to map the object operation and the section in the source code that is executed when the operation occurs.

We have to determine the granularity of a source code section to be recorded. The recording unit in the current version of Kanon is *sentences*. Given this implementation, our approach regards AST nodes as sections.

We consider the program shown in Fig. 12 for explanation. A description of `fldA: 100` in Line 5 is the source code section corresponding to an object initialization. We record this section together with the initialization operation.

3.1.3 Call Stack

In some cases, Kanon and other live data structure programming environments have to know object identities before and after program editing; Kanon uses a call stack for identification. In particular, when a function call occurs, Kanon records a source code section that calls the function and stores it as a *call stack*.

Given the program shown in Fig. 12, in Line 7, the function `f` defined by Lines 1 and 3 is called. Note that this function manipulates the object taken as an argument in Line 2. When this operation is performed, the call stack has only one element that is the function call on Line 7. Our approach also records the call stack when the operation in Line 2 occurs.

3.2 Extraction of Object History with Language Implementation Framework

This section describes a method to obtain information that is necessary to construct an object graph during program execution by using a language implementation framework.

3.2.1 Recording Object Operation

To capture a run-time object operation, we patch a framework and an interpreter. Moreover, we design this patch to notify the framework that an object operation happens. Object operation occurs when the interpreter executes a source code section that manipulates an object. The source codes that perform object manipulation are written in an appropriate area in the interpreter. Our approach inserts a process to notify the side of a framework of information about an object operation after the operation occurs. Moreover, we add functions to collect operation notifications to the framework.

Next, we define the sender technique that throws notifications to systems that have an object graph construction mechanism.

3.2.2 Source Code Section

Our system monitors the entire execution of an AST node. After noticing all nodes are executed, it links each object operation to each section. For example, we assume the situation whereby there are two nodes (A and B), and node B is being executed after executing node A. In this situation, this technique allows us to count all object operations after executing node A that belong to node B.

We explain the detail of this design as follows. The system first prepares an empty buffer and adds information to it when an object operation occurs. When the execution of a node is finished, the system links the object operations in the buffer with the node.

3.2.3 Call Stack

The system monitors the execution start and end of the function call node to maintain the call stack virtually. When an object operation occurs, it links the call stack with the operation.

3.3 Constructing an Object Graph with Intermediate Representation Generated from Object Operation History

In this section, we describe the concrete design of a IR we propose to solve the problem of slow construction using our operation history technique.

Table 1 Correspondence table from positions in source code to time.

Line	Array at a time
2	[2]
5	[1]
6	[3]
7	[4]
8	[5]

```
1 function example(o) {
2     obj.bar = 100;          // Time t = 2
3 }
4
5 let obj = { foo: "hello" }; // Time t = 1
6 example(obj);               //      t = 3
7 Console.log(1);             //      t = 4
8 obj.bar = "baz";            //      t = 5
```

Fig. 13 Example of JavaScript program.

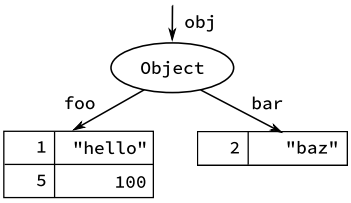

Fig. 14 Hashtable of object operations.

3.3.1 Time-Series Method

Here, we propose the *time-series method*, which records object operations during program execution and generates an object graph at a certain point of the execution on demand.

First, this method manages each running process *by time*. Specifically, it uses a global time variable for time management. When an AST node's `execute` method finishes, we take the value of a current time variable as the finish time of that node. For all AST nodes, we collect their finish times to get a correspondence between time and source code sections. Thus, when the user points the cursor at a certain section on the source code, the method can obtain the time of execution of the section. For example, we can get the correspondence shown in **Table 1** from the program shown in **Fig. 13**.

Next, this method creates a hash table. The key is a couple of an object and a field identifier, and the value is an array whose element is a pair of a time and its value. In particular, this method monitors and collects when the execution of an AST node ends. When an object is generated, it gathers all referable fields and inserts their times and values into the hash table at an appropriate position. In the case of the program shown in Fig. 13, we can obtain the records shown in **Fig. 14** from the program.

When the user sends a request for visualizing an object graph, the system generates it using the following procedure:

(1) It collects a time when a section specified in the source code.
(2) It searches the hash table to generate the object graph at that time. Specifically, it first selects an object to begin scanning, such as a global variable. Next, it examines all of its fields and records the value if it is a primitive value, such as a number or a string, at that time. If the value is a reference to the other object, it records this reference and examines the object. The system repeats this procedure until it reaches a primitive value and then produces an object graph at that

time.

(3) It caches the constructed object graph with the time. If the user requires an object graph that has already been constructed, the system returns the cached one.

We describe the time complexity of the conversion to the IR during the execution and that of the construction of the object graph in this method. Hereinafter we assume that the hash table and the array have an ideal implementation. First, the execution cost is $O(M)$, as in a naïve technique using operation history, where M is the total number of the object operations. Next, we consider the cost of the construction. By binary search, we can obtain a value of a particular field in an object at a certain time. The cost of the construction is $O(L \log N)$, where L is the total number of fields and N is the total number of objects. Therefore, the time-series method seems to construct the object graph faster than the naïve method when the number of object operations is large.

4. Implementation Using Graal/Truffle and GraalJS

This section describes how to implement our proposals from Section 3. To be specific, we show the implementation design by using Graal/Truffle that records and extracts operation history collection at run-time.

4.1 Recording Object Operations Used for Object Graph Collection

We define the data representation of an object operation as shown in **Fig. 15**. Hereinafter, `object` denotes an object to be operated, `key` denotes an identifier of the field, and `value` denotes a value to be assigned.

4.2 Extraction of Object Operation History Using Graal/Truffle

4.2.1 Object Operation History

We define a class `ObjectTracker` and an interface `ObjectChangeListener` in the Truffle framework (as shown in **Fig. 16**). The class `ObjectTracker` receives object operation notifications from the interpreter and notifier systems that require operation histories. The interface `ObjectChangeListener` is a listener for receiving operation notifications. Both have a method `onFieldAssigned`. The system that requires the operation history registers the implementation of `ObjectChangeListener` with `ObjectTracker`. When the interpreter manipulates an object, it calls the method `onFieldAssigned` of `ObjectTracker` to notify `ObjectTracker` of the content of the operation. When the method `onFieldAssigned` of `ObjectTracker` is called, it calls that of each registered `ObjectChangeListener` to notify them of the operation. These implementations enable the framework to send information about the object operation to the system.

We select a subclass of `SetCacheNode` as an example to explain the implementation described above. A class `SetCacheNode` is a superclass that performs an object operation in the target language. Depending on the type of object to be manipulated and the value to be assigned, there are multiple subclasses of `SetCacheNode`. The class `SetCacheNode` has an abstract method `setValue`, which takes an instance of `DynamicObject` to be operated on and a value to be assigned as arguments. It performs the operation based on the information of the subclass. We show a class `ObjectPropertySetNode` as an example subclass of `SetCacheNode` in **Fig. 17**. It is specialized to manipulate an instance of `DynamicObject` and has an instance of `Property` as a member. The method `setValue` of `ObjectPropertySetNode` updates the value of the field of the given `DynamicObject`, which the member variable of type `Property` represents, to the passed one (as shown in Line 5).

```
1 public class ObjectChangeEvent {
2     Object object;
3     Object key;
4     Object value;
5 }
```

Fig. 15 Definition of a recorded object manipulation.

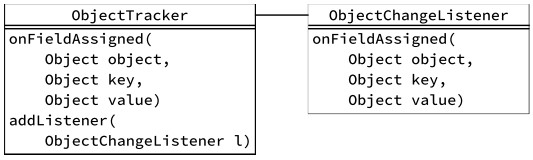

Fig. 16 Class diagram of `ObjectChangeListener` and `ObjectTracker`.

```
1  class ObjectPropertySetNode extends SetCacheNode {
2      Property property;
3
4      void setValue(DynamicObject store, Object value)
           {
5          property.set(store, value);
6      }
7  }
```

Fig. 17 Example of definition of a subclass of `SetCacheNode`.

```
1  class ObjectPropertySetNode extends SetCacheNode {
2      Property property;
3
4      void setValue(DynamicObject store, Object value)
           {
5          property.set(store, value);
6          TruffleFramework.ObjectTracker.
               onFieldAssigned(
7              store,
8              property.getKey(),
9              value);
10     }
11 }
```

Fig. 18 Example involving modification for notifying `SetCashNode` of object updates.

Figure 18 shows the implementation of `ObjectPropertySetNode` modified in the way explained above to notify the framework of an object change. Line 5 updates the object, and Lines 6 and below send the operation to the framework.

We have added 397 lines to Truffle to implement the proposed process. We have also inserted 169 lines and changed 133 lines to GraalJS. We mention that the total number of lines of code in Truffle and GraalJS containing our modification is approximately 1.70 million and 265 thousand, respectively.

4.2.2 Source Code Section

This section details the implementations that monitor the execution of AST nodes of the target language on Graal/Truffle.

We use the `ExecutionEventNodeFactory` and `ExecutionEventNode` defined in the Instrumentation API to record the end

of the execution of each node. When an object operation occurs, they obtain the section in which the execution is completed first after the operation. Here, we have to consider the nullability of the value that the method `getSourceSection` of `Node` returns. Given this, we implement recording of the non-null section.

4.2.3 Call Stack

We also use the `ExecutionEventNodeFactory` and `ExecutionEventNode` to construct a call stack. As we explained in Section 2.2.3, `ExecutionEventNodeFactory` can specify which nodes are monitored by `ExecutionEventNode`. Truffle provides the tag `StandardTag.CallTag`, which shows that it calls a function. Thus we can record the start and end of the node that is labeled by `StandardTag.CallTag` to obtain a call stack.

5. Evaluation

5.1 Targets of Evaluation

In this section, we evaluate the performance of the snapshot technique and the operation history technique by comparing the following methods:

- original implementation of Kanon (Original),
- naïve method using operation history (Naïve),
- time-series method (Time-series),
- copy method (Copy) (we describe this in Section 5.1.1).

The original implementation of Kanon converts the given program into one with checkpoints and executes it to construct the object graph. Moreover, it generates an object graph each time the checkpoint is executed. By contrast, the proposed method does not perform the program conversion, since the mechanism of constructing object graphs is implemented in the framework and the interpreter. While the copy method generates an object graph each time the checkpoint runs, the time-series method generates it just-in-time. We have evaluated these and considered how their differences affect performance.

We have also assessed the performance of the execution in bare GraalJS (no instrumentation) to measure the overhead of the generation of snapshots, the collection of operation history, and the construction of object graphs.

5.1.1 Copy Method

We imitated the snapshot technique in our system to compare the two techniques: the original Kanon and our operation history technique. In the snapshot technique, a system initializes an empty graph. When the object operation occurs, it copies the current graph generated by the last object modification. Then it reflects the new operation to the duplicate and stores it with the current time. These processes generate the time series of the object graphs. When the user requires the graph at a certain section of the source code, it converts the section into time and returns the latest graph before the time. It can obtain the latest one by binary search.

Using this technique, the object graph shown in **Fig. 19** is obtained from the program shown in Fig. 13. When the system executes the program, the object operations performed at Times 1, 2, and 5 are recorded. When the object graph at the end of Line 3 is needed, it returns the graph at Time 2, since it is the latest one at Time 3.

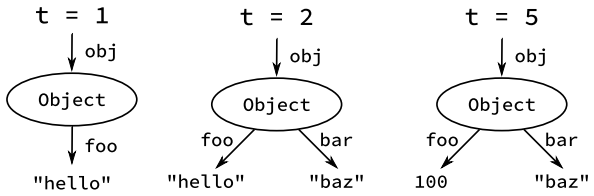

Fig. 19 Object graphs at each time.

5.2 Evaluation Environment

The environment for this evaluation is as follows.

- AMD Ryzen 9 5900X,
- DDR4-2666 64GB RAM,
- Linux 5.10.61-gentoo,
- OpenJDK 1.8.0_292 / 11.0.11_p9-r1,
- V8 9.5.151.

5.3 Evaluation Method

For the original Kanon and our proposed implementations, we carried out two performance measurements, as explained in Section 5.3.1 and Section 5.3.2. For each program, we iterated 30 times and calculated the mean and standard error.

To consider the differences of interpreters, we executed the original Kanon on GraalJS and V8 and measured its performance. V8 [12] is a JavaScript engine used in some browsers, such as Chromium.

5.3.1 Measuring the Execution Time

We measured the execution time to obtain the object operation history for each target. In particular, we assess the time required for the following processes:

- for the original implementation: the conversion of the program and its execution;
- for the time-series and the copy methods: the execution of the program with recording of the object operations in the proposed manner.

We executed these processes enough times to perform a warmup that is not included in these measurements.

5.3.2 Measuring the Construction and Selection Time of Object Graphs

We measured the time required for object graph selection and construction when the user requests the graph at the end of the last line of the program. For the same reason as for warmup times in Section 5.3.1, we executed the program enough times but this is not used in the measurement.

5.3.3 Test Programs for the Measurement

To perform the above performance evaluation, we used three programs that add elements based on a specific sequence to each of the data structures of an array-based heap, a doubly linked list, and an AVL tree.

5.4 Evaluation Results

5.4.1 Measurement Result of the Execution Time

Since each program to be evaluated inserts elements repeatedly, the number of object operations is considered to be proportional to the size of the data structure. Indeed the number of object operations increased in proportion to the size of the data structure, as shown in **Table 2**, **Table 3**, and **Table 4**. **Figure 20**,

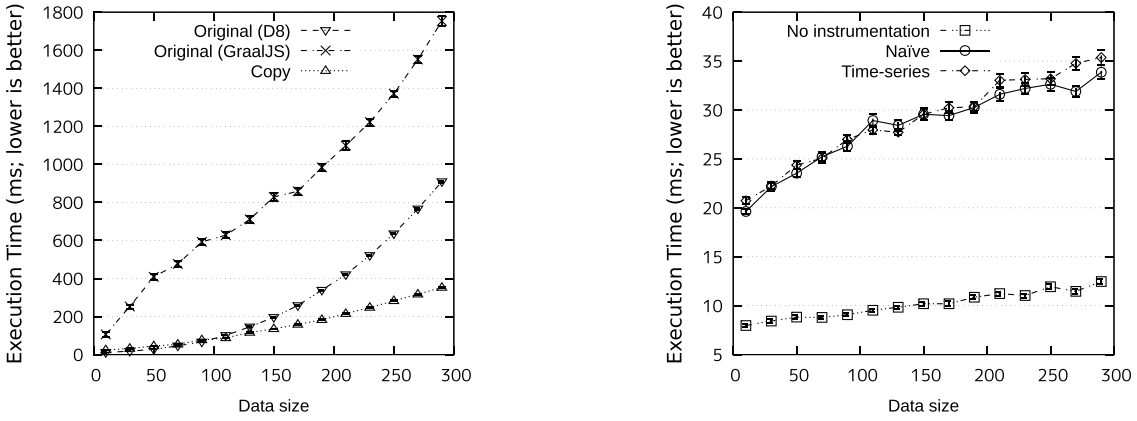

Fig. 20 Comparison of execution time on DLList with each method.

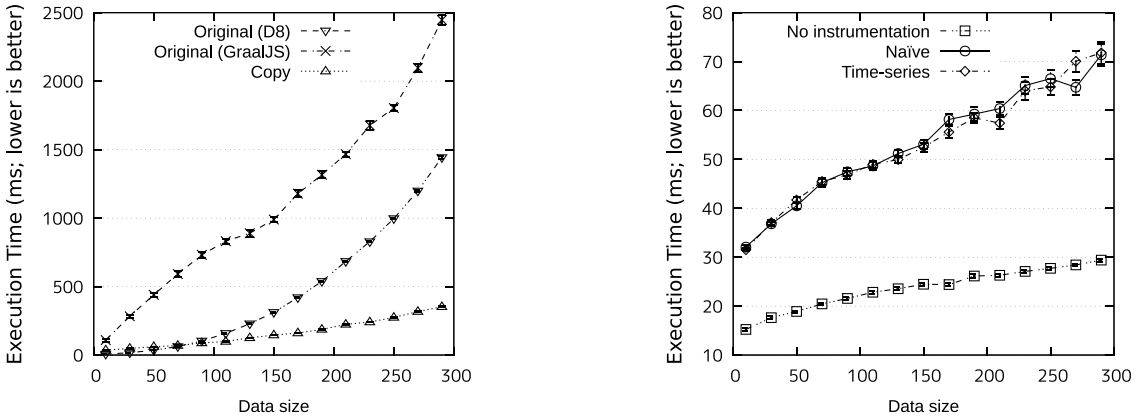

Fig. 21 Comparison of execution time on Heap with each method.

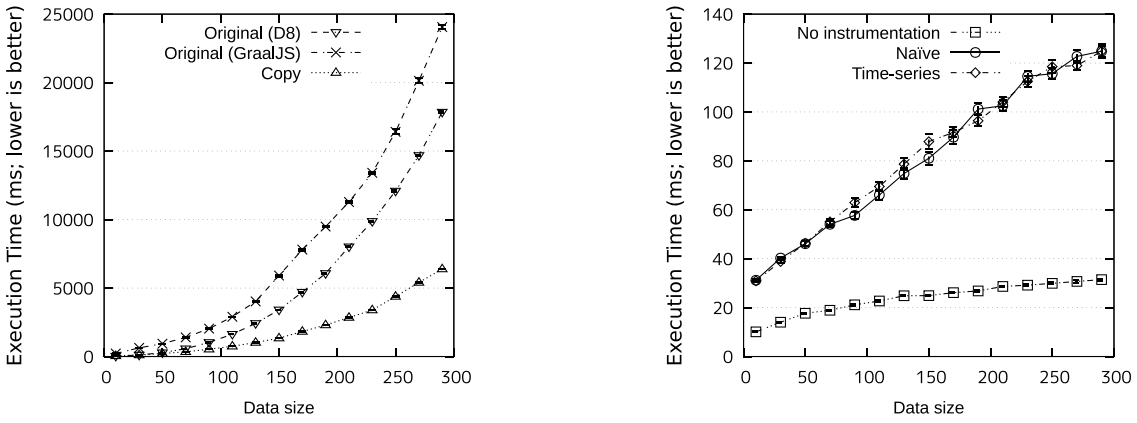

Fig. 22 Comparison of execution time on AVL Trees with each method.

Fig. 21, and **Fig. 22** are measurement results for the execution time of each program. **Table 5** explains the legends in the evaluation results.

The time complexity in conventional implementation and that in the copy method seem to be $O(N^2)$, where N is the size of the data structure. The execution in the copy method was faster than that in the original implementation. On the other hand, the execution in the naïve method with object history and that it the time-series method were performed in $O(N)$, and the slope and the intercept of both were approximately equal.

5.4.2 Measurement Result of the Construction and Selection of an Object Graph

Figure 23, **Fig. 24**, and **Fig. 25** show the time required to construct or select an object graph. Since the system simply selects the proper graph from the list of constructed object graphs in the copy method, it was the fastest of the three methods. Whereas the theoretical time complexity is considered to be $O(\log M)$, the selection was carried out in constant time in this measurement. The generation times in the naïve method and in the time-series method for the program using the doubly linked list were proportional to N, where N is the size of the data structure. In contrast, for the other programs, those in the naïve method were proportional to N, but those in the time-series method were constant or proportional to N.

Table 2 Number of object operations in DLList.

Data size	Number of operations
10	193
30	553
50	913
70	1273
90	1633
110	1993
130	2353
150	2713
170	3073
190	3433
210	3793
230	4153
250	4513
270	4873
290	5233

Table 3 Number of object operations in Heap.

Data size	Number of operations
10	228
30	913
50	1766
70	2675
90	3675
110	4675
130	5696
150	6836
170	7976
190	9116
210	10256
230	11396
250	12536
270	13781
290	15061

Table 4 Number of object operations in AVL Tree.

Data size	Number of operations
10	717
30	2852
50	5299
70	7850
90	10600
110	13350
130	16109
150	19119
170	22129
190	25139
210	28149
230	31159
250	34169
270	37344
290	40614

Table 5 Explanation for the legends in the evaluation results.

Legend	Description
Original (D8)	Execution of original Kanon on D8
Original (GraalJS)	Execution of original Kanon on GraalJS
Naïve	Execution in the naïve method using operation history
Copy	Execution in the copy method (introduced in Section 5.1.1)
Time-series	Execution in time-series method
No instrumentation	Execution on GraalJS without any instruments

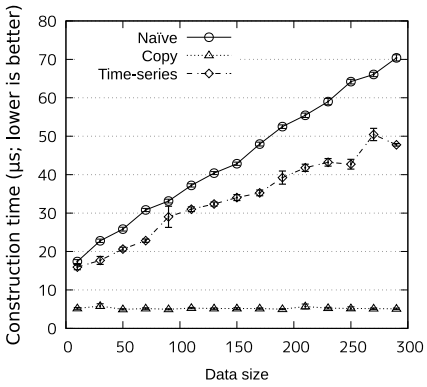

Fig. 23 Snapshot acquisition and construction time in DLList.

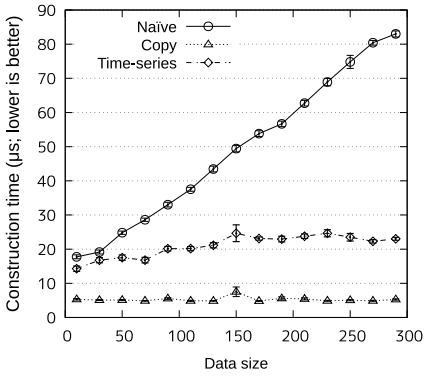

Fig. 24 Snapshot acquisition and construction time in Heap.

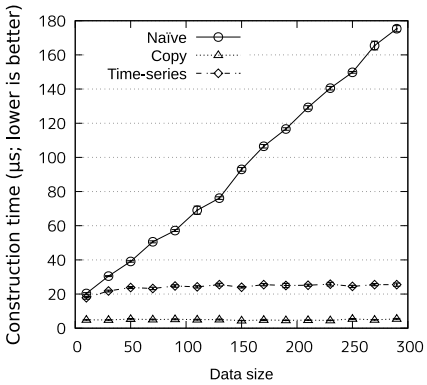

Fig. 25 Snapshot acquisition and construction time in AVL Tree.

6. Related Work

6.1 Babylonian System

Babylonian Programming System (BPS) [13] is a language-agnostic live programming environment. It is also implemented using Graal/Truffle. **Figure 26** shows a screenshot where it analyses the JavaScript program via Visual Studio Code. It provides some features, such as obtaining and displaying the value returned by the function and arguments specified by the user. That is, it allows users to write the program and debug it simultaneously. BPS obtains the runtime information with GraalVM similarly to our proposed method. In addition, it performs static analysis to collect information that is not available from the current API of GraalVM. On the other hand, we proposed to slightly modify the Truffle framework in order to obtain on it all of the necessary information required to construct object graphs.

6.2 OnlinePythonTutor

OnlinePythonTutor [14] is also a live programming environment specialized in data structures. It executes the given program and displays the graph based on reference relations of objects

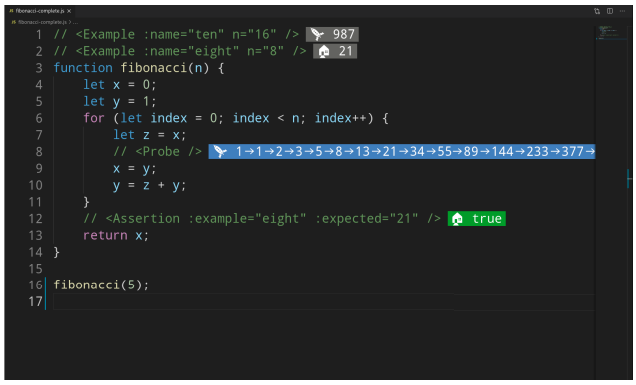

Fig. 26 Babylonian Programming System running on VSCode.

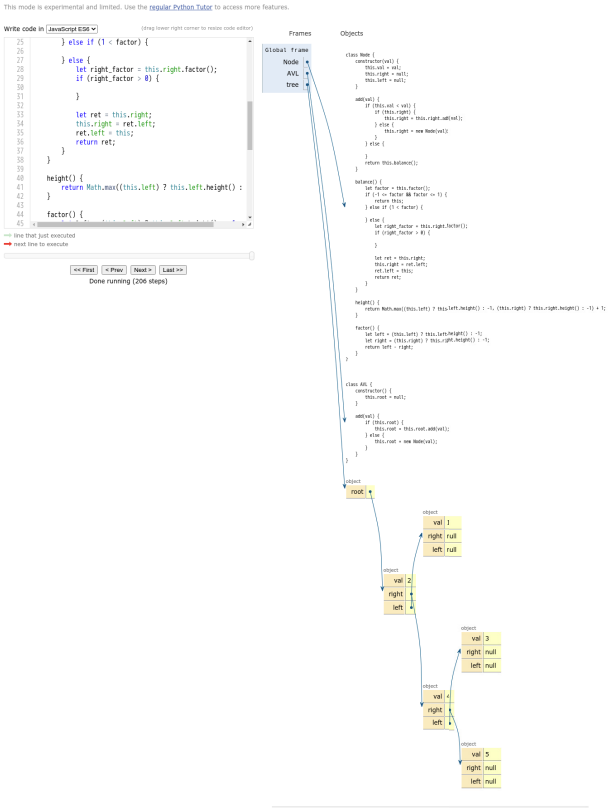

Fig. 27 Screenshot of Online Python Tutor running a program using an AVL tree.

constructed during the execution (**Fig. 27**), similarly to Kanon. It accepts programs written not only in Python but also JavaScript. In OnlinePythonTutor, its editor and the parts that display the graph are common to all languages, while the parts collecting the object graph are implemented from scratch for each language.

7. Conclusion

In this paper, we proposed recording the history of object constructions and modifications for live programming environments specialized to data structures to collect object graphs. We also suggested implementing it by extending a VM on the language implementation framework. By extending Graal/Truffle, we implemented a mechanism for the collection of object graphs that are the same as those obtained by Kanon. In addition, we measured the execution time and construction time of the object graph for the original implementation and proposed method. We concluded that the proposed method improves performance.

In future work, we plan to connect the implementation on Graal/Truffle to the original Kanon. We have already implemented the parts that obtain the information required by Kanon, but now we have to connect the current Kanon with the proposed method seamlessly to generate node-link diagrams based on this implementation.

We must also generalize the proposed method, and we also plan to apply it to languages other than JavaScript. In particular, in the VMs with Graal/Truffle, the basic operations such as object generation seem common to all languages. Thus, we expect that it will be possible to apply our method to other languages without substantial changes.

In addition, enabling live programming environments such as Kanon to be usable in any development environment is the object of further study. Since many of them now support the language server protocol (LSP) [15], we consider implementing Kanon using LSP to be valuable.

References

[1] Tanimoto, S.L.: A Perspective on the Evolution of Live Programming, *Proc. 1st International Workshop on Live Programming*, *LIVE '13*, pp.31–34, IEEE Press (2013).

[2] Maloney, J., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E.: The Scratch Programming Language and Environment, *ACM Trans. Comput. Educ.*, Vol.10, No.4 (online), DOI: 10.1145/1868358.1868363 (2010).

[3] Kato, J., McDirmid, S. and Cao, X.: DejaVu: Integrated Support for Developing Interactive Camera-Based Programs, *Proc. 25th Annual ACM Symposium on User Interface Software and Technology*, *UIST '12*, pp.189–196, ACM (online), DOI: 10.1145/2380116.2380142 (2012).

[4] Aaron, S. and Blackwell, A.F.: From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages, *Proc. 1st ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, *FARM '13*, pp.35–46, ACM (online), DOI: 10.1145/2505341.2505346 (2013).

[5] Oka, A., Masuhara, H. and Aotani, T.: Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming, *Proc. 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, *Onward! 2018*, pp.72–87, Association for Computing Machinery (online), DOI: 10.1145/3276954.3276962 (2018).

[6] Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D. and Wimmer, C.: Self-Optimizing AST Interpreters, *SIGPLAN Not.*, Vol.48, No.2, pp.73–82 (online), DOI: 10.1145/2480360.2384587 (2012).

[7] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D. and Wolczko, M.: One VM to Rule Them All, *Proc. 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, *Onward! 2013*, pp.187–204, Association for Computing Machinery (online), DOI: 10.1145/2509578.2509581 (2013).

[8] Archambault, D. and Purchase, H.C.: The mental map and memorability in dynamic graphs, *2012 IEEE Pacific Visualization Symposium*, pp.89–96 (online), DOI: 10.1109/PacificVis.2012.6183578 (2012).

[9] Bolz, C.F., Cuni, A., Fijalkowski, M. and Rigo, A.: Tracing the Meta-Level: PyPy's Tracing JIT Compiler, *Proc. 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, *ICOOOLPS '09*, pp.18–25, Association for Computing Machinery (online), DOI: 10.1145/1565824.1565827 (2009).

[10] Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D. and Grimmer, M.: Practical Partial Evaluation for High-Performance Dynamic Language Runtimes, *SIGPLAN Not.*, Vol.52, No.6, pp.662–676 (online), DOI: 10.1145/3140587.3062381 (2017).

[11] Wöß, A., Wirth, C., Bonetta, D., Seaton, C., Humer, C. and Mössenböck, H.: An Object Storage Model for the Truffle Language Implementation Framework, *Proc. 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, *PPPJ '14*,

pp.133–144, Association for Computing Machinery (online), DOI: 10.1145/2647508.2647517 (2014).
[12] Project, V.: V8 JavaScript engine, available from ⟨https://v8.dev/⟩.
[13] Niephaus, F., Rein, P., Edding, J., Hering, J., König, B., Opahle, K., Scordialo, N. and Hirschfeld, R.: Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM, *Proc. 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, *Onward! 2020*, pp.1–17, Association for Computing Machinery (online), DOI: 10.1145/3426428.3426919 (2020).
[14] Guo, P.J.: Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education, *Proc. 44th ACM Technical Symposium on Computer Science Education*, *SIGCSE '13*, pp.579–584, Association for Computing Machinery (online), DOI: 10.1145/2445196.2445368 (2013).
[15] Microsoft: Official page for Language Server Protocol, available from ⟨https://microsoft.github.io/language-server-protocol/⟩.

Shusuke Takahashi received his B.S. degree in Computer Science from Tokyo Institute of Technology in 2021. He is currently a master student at Department of Mathematical and Computing Science, Tokyo Institute of Technology.

Yusuke Izawa is a doctoral student at Department of Mathematical and Computing Science, Tokyo Institute of Technology. He received his B.S. and M.S. degrees from Tokyo Institute of Technology in 2018 and 2020, respectively. His research interests include design and implementation of programming languages, program optimization, and programming experience.

Hidehiko Masuhara is a Professor at Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his B.S., M.S., and Ph.D. degrees from the University of Tokyo in 1992, 1994 and 1999 respectively. Before joining Tokyo Institute of Technology, he served as an Assistant Professor, Lecturer, and Associate Professor at Graduate School of Arts and Sciences, the University of Tokyo. His research interests include design and implementation of programming languages and software development environments.

Youyou Cong is an assistant professor of the Department of Mathematical and Computing Science, Tokyo Institute of Technology. She received her Ph.D. from Ochanomizu University in 2019. Her research is centered on the theory and applications of delimited continuations and type systems.