JavaScript Debugger Using Data Structure Visualization (JavaScript のデータ構造可視化を用いたデバッガ)

東京工業大学大学院
情報理工学研究科数理・計算科学
学籍番号：12M54060
徐駿剣

2013 年度修士論文
指導教員 脇田 建 准教授

July 15, 2014

Abstract

Debugger is used to test and debug target programs by stepping through the program and examining the current program state by evaluating the values of expressions or checking the stack trace. However, the character-based expressions evaluation provides limited insight into the current program state. This paper examines how a debugger can provide a higher-level, more informative visualization based on data structure. Except static view, the debugger also allows generating animation while target object is modified. However, high-level visualization typically relies on user augmented source code. This paper enables the debugger to understand the data structure of target object by externally supplied semantic information which is written in a declarative language called visualjs.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Program visualization systems translate program into visual shapes. It is often used in algorithm animation systems where algorithm behavior is visualized by producing the abstraction of the data and the operations of the algorithm. Such visualization also can be used in a debugger, as it allows for better understanding on the behavior of the program and help perceive which parts of the program does not function correctly.

A debugger is used to test and debug target programs. It offers many sophisticated functions such as running a program step by step, pausing the program at some event or specified instruction by means of a breakpoint, and examining the current state by evaluating the values of expressions or checking the stack trace. However, the character-based expressions evaluation provides limited insight into the current program state. Modern IDEs intend to solve this problem by offering a dedicated view to watch expressions. As a typical example of object-oriented programming language, when the variable being watched is of reference type, its fields is listed in a tree-view table. The root of this list represents the value itself. If a field is also of reference type, it can be further expanded in the same manner. Fields of primitive type can not be further expanded.

However, such visualization still has deficiencies. Firstly, all of the objects are visualized in a tree-view table, and hence lack of important semantic information. Secondly, The whole view will be refreshed while stepping through the program. It is nearly impossible to check the modification parts and find the relevance between two succeeding states.

1.2 Main Contribution

This paper examines how a debugger can visualize data structure and generate smooth animation automatically while according object is modified. The user has to write VisualJS, a new declarative language describing semantic information of the data structures, by himself. Instead of displaying object in a general way, we wish to produce more helpful and informative visualization.

1.3 Outline of the Thesis

The structure of the thesis is as follows. Chapter 2 gives an introduction to the visual debugging and program visualization as related research. We review a series of typical systems and compare them with our research. In Chapter 3, we introduce the design of a new declarative language called VisualJS, and explain how to make use of VisualJS to establish the mapping relationship between a object graph and visual representations. In Chapter 4, we discuss the implementation details of the system. In Chapter 5, we evaluate the system via several examples. Finally, Chapter 6 concludes the thesis including a discussion of possible future work.

Chapter 2

Related Research

This chapter introduces several related research, and compares these research with ours.

2.1 Debuggers

In general, the purpose of using debuggers is to detect the existence of errors in a program, to locate their position or cause, and, finally fix them [10]. Several typical debuggers are introduced as follows.

2.1.1 GDB

The GNU Debugger [3], usually called GDB, is a debugger written in C and C++. Its usage is not strictly limited to the GNU operating system. It could run on many Unix-like systems for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Java and so on.

GDB has not its own graphical user interface, but defaults to a command-line interface. Users have to remember various commands like stepping through, setting breakpoint, evaluating expressions and so on. However, character-based interaction lacks of insight into the target program.

To solve the problems of command-line interface, debuggers are usually integrated into a comprehensive development environment, usually called IDE, to maximize programmer productivity by providing tight-knit components with an integrated user interface, such as Eclipse, IntelliJ IDEA, NetBeans, and Microsoft Visual Studio.

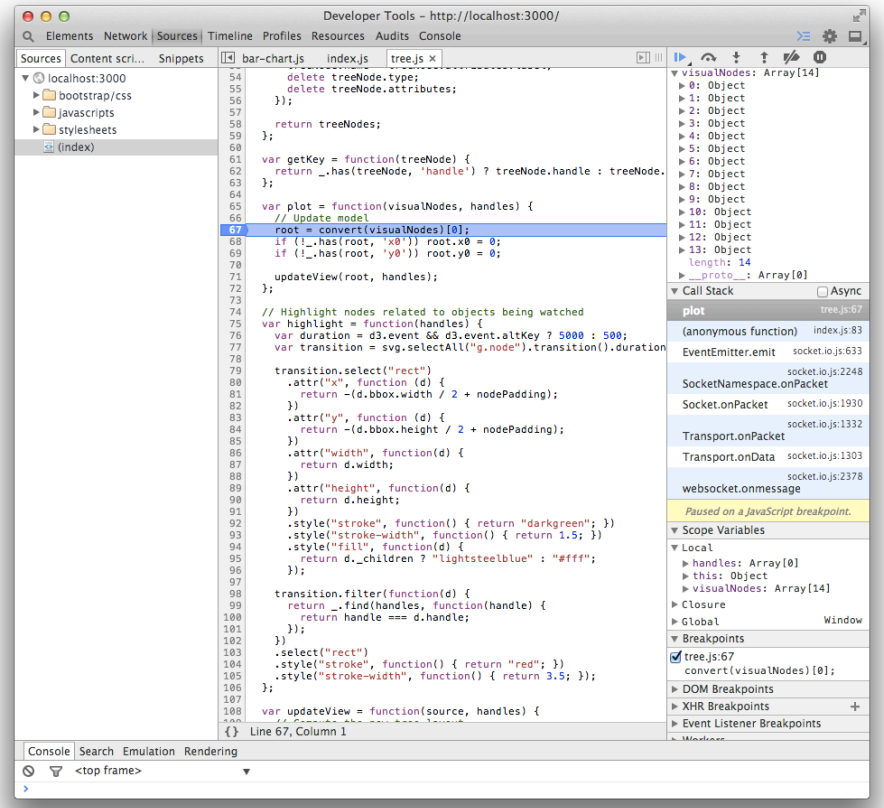

Figure 2.1: A screenshot of debugging JavaScript code in chrome developer tools

2.1.2 Chrome Developer Tools

The chrome developer tools [1], also called devtools, are a set of web authoring and debugging tools built into google chrome. Figure 2.1 is a screenshot of chrome developer tools when debugging JavaScript code.

Devtools include a number of useful tools to help debug JavaScript. The screenshot shows the sources panel providing a graphical interface to the V8 debugger. The sources panel shows all the code of the inspected page. We can see standard controls to pause, resume, and step through program on the right side. Other runtime information such as watched variables, call stack, scope variables, and breakpoints are also visible one the right side below the standard controls.

Our system provides users with similar debugging experience of devtools. However, devtools can not provide deeper insight into the runtime infor-

mation in that all these information are still character-based. Our research intends to help programmers better understand the program via the visualization of data structure.

2.2 Visual Debuggers

Visual debuggers are tools that reflect code-level aspects of program behavior, showing execution proceeding statement by statement and visualizing the stack frame and the contents of variables and are directed more toward program development rather than understanding program behavior [21]. Visual debuggers usually let users execute the program in steps while allowing them to simply understand the flow of data.

2.2.1 DDD

GNU DDD (Data Display Debugger) [2] is a graphical front-end for command-line debuggers such as GDB described in Chapter 2.1.1. Besides usual front-end functions such as viewing source code, DDD also allows interactive data display. Data structures can be displayed as kinds of graphs like Figure 2.2.

2.2.2 jGRASP

jGRASP (Graphical Representation of Algorithms, Structures, and Processes) [9, 17, 8, 7] is a lightweight development environment, created specifically to provide various visualizations to improve the program comprehension: syntax highlighting, control-structure-diagram, UML class diagrams and object viewers. jGRASP is implemented in Java and support for Java, C, C++, Objective-C, Python, Ada, and VHDL, but most advanced features are only available for Java.

jGRASP object viewers are tightly integrated with the debugger. It can be used to automatically detect the data structure such as linked lists, binary trees, and array wrappers (lists, stacks, queues, etc.). For linked structures, the visual representations can be animated to show nodes being added and deleted form the data structure. Figure 2.3 shows two ArrayLists being visualized.

2.2.3 Online Python Tutor

Python Tutor [15] is a web-based program visualization tool for Python. Using this tool, teachers and students could write Python programs directly

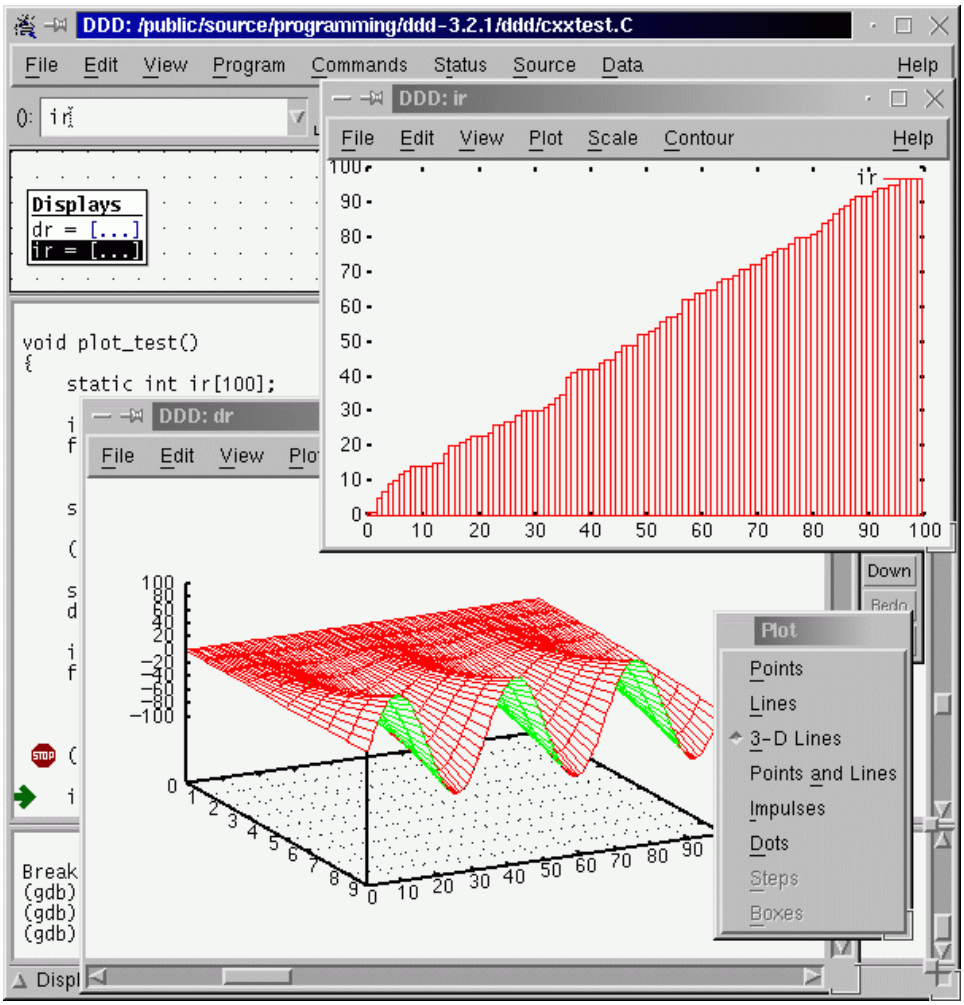

Figure 2.2: A screenshot of DDD visualizing an one-dimensional array and a two-dimensional array

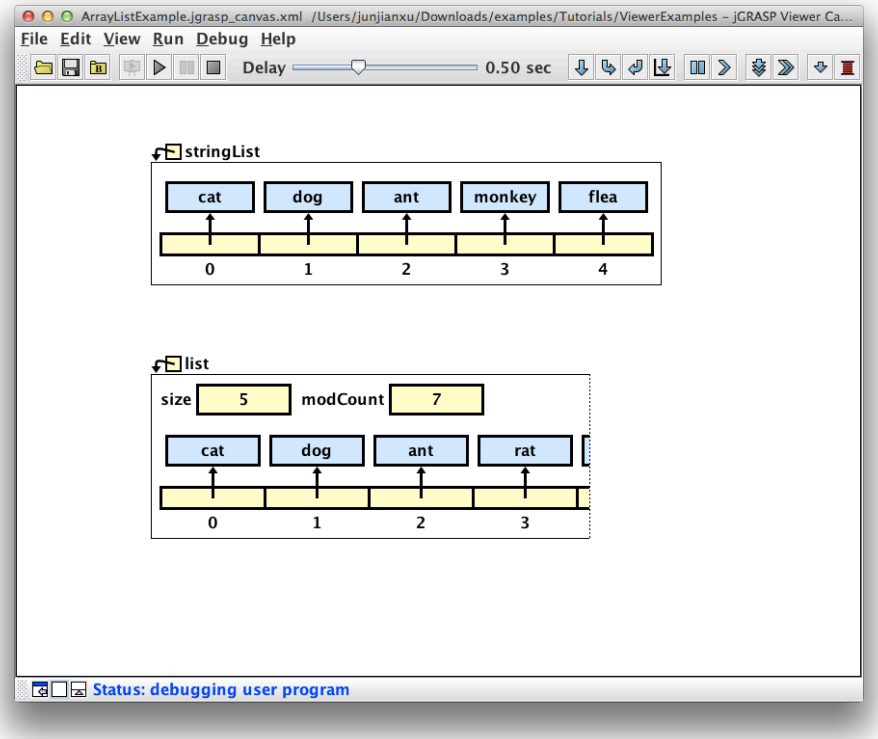

Figure 2.3: A screenshot of jGRASP object viewer

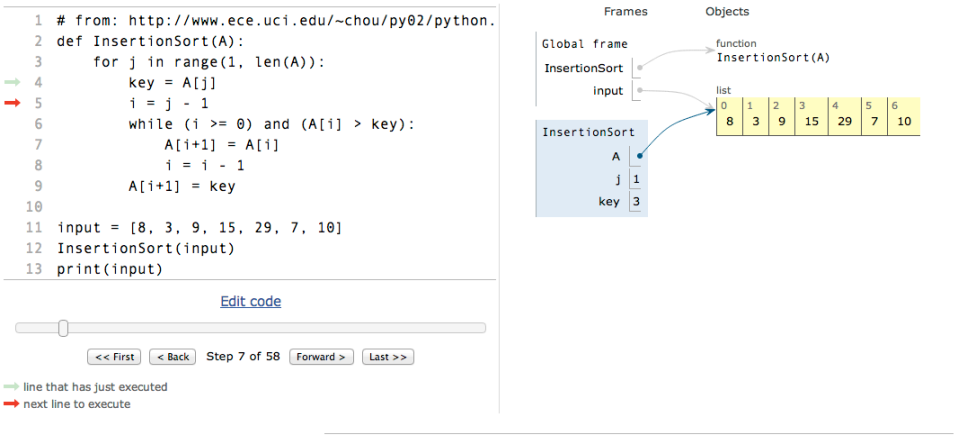

Figure 2.4: A screenshot of Python Tutor visualizing an insertion sort program

in the web browser, step forwards and backwards through execution to view the run-time state of data structures. Figure 2.4 shows an example of visualizing an insertion sort program. Users can:

1. view the currently executing source code
2. step forwards and backwards through execution
3. view stack frames and variables
4. view heap object contents and pointers
5. view the program's text console output
6. generate a sharable URL of the current visualization

2.2.4 JIVE

JIVE (Java interactive software visualization environment) [13, 14] is a Java program visualization tool integrated within the Eclipse IDE. JIVE proposed a new notation to display runtime object structures as the environments of program execution. The object structures can be displayed in multiple views with different granularities, allowing the user to focus on his interest. JIVE shows the execution history with dynamically generated sequence diagram like those used in Unified Modeling Language. JIVE supports for forward and backward execution so that bugs can be reproduced. JIVE supports queries on the runtime state. A user can inquire of the system of when variables have changed, when variables have held certain values, when objects

are created, and when methods are called. JIVE automatically generates clear layout for all diagrams.

2.2.5 Jype

Jype [16] is an educational software tool to help students learn elementary programming. Unlike most existing systems focusing on a single aspect of learning. Jype implemented a combined system that closely integrates programming teaching with visualizations tools to help students understand how programs works on a conceptual level, and automatic assessment tools to give feedback on submitted assignments.

2.3 Algorithm Animation

The program state changes as stepping through the execution so the static visualization is not fixed any more. Instead of static visualization, we can use algorithm animation, a sequence of images which are shown on after another [10], here. The animation would better be smooth to help programmers understand the transition between two execution points.

2.3.1 JSAV

JSAV (JavaScript algorithm visualization) [18] is a JavaScript framework for creating engaging algorithm visualizations with active learning features. It is meant to be used in any web browser like Python Tutor described in Chapter 2.2.3. JSAV supports different engagement levels. The simplest way is static visualization of data structures. Another way is called slideshows, which shows a series of steps of the algorithm animation. Figure 2.5 shows an example of a JSAV visualization.

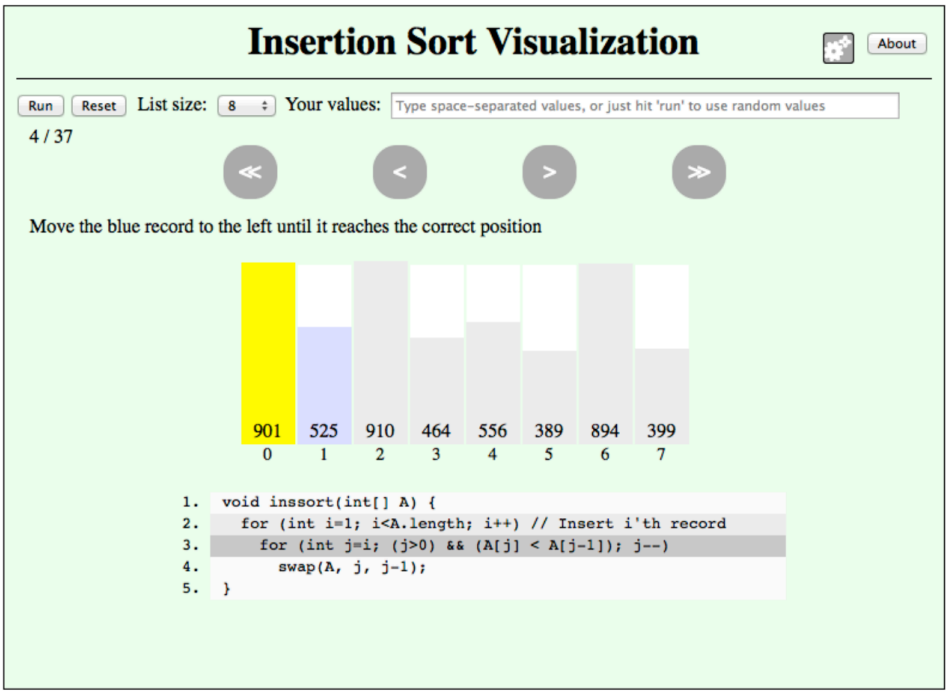

Figure 2.5: A screenshot of JSAV visualizing an insertion sort program

Chapter 3

System Proposal

To solve the problems mentioned in last chapter, our system used traversal-based method [20] to traverse the object. A new declarative language called VisualJS is created so that this approach can be oriented to JavaScript. On top of this, we proposed a mapping mechanism from an object to visual shapes. Making use of the mapping relationship, our system is able to visualize the object in kinds of shapes and animate these visual shapes automatically while stepping through program. Note that instead of providing rich kinds of data structure visualizations, our research intends to help users visualize and animate the objects that they are interested in.

3.1 The Design of VisualJS

The traversal-based approach takes a root object and traverse the objects by following any other objects referenced by the root. A set of predicates aligned in a pattern will be applied to the object and decide which action to execute. Each action describes its rules about how to generate visual nodes and which objects to traverse as follows.

The declarative language is written in text file. Each file describes only one data structure, which means you have to write code for objects with different structures in respective files. Each file is made up of one pattern and several actions.

The guiding principles that were considered when we designed this system are as follows:

1. **Expression Power:** The topology relationship of visual shapes should be specified clearly, naturally, intuitively, and concisely.

2. **Flexibility:** Flexibility provides the design with root to extend. More data structures and sophisticated visual attributes would be supported in the future. All these requirement changes should be considered in initial design.

3.1.1 Data Model

Although this system is constructed based on JavaScript, it theoretically suits all object-oriented programming languages like C++ and Java and any other programming language in which an object is constructed in a recursive way, which means an object is composed of other primitive type values or objects. Hence the object can be traversed and matching actions can be executed to generate the topology of visual nodes.

3.1.2 Pattern

A pattern aligns a series of predicates to be matched when object comes. Each predicate has its pair action. The first action whose predicate is matched will be executed. We will elaborate pattern by showing its grammar in EBNF as follows:

⟨*pattern*⟩ ::= pattern-name **: pattern {** ⟨*pattern-body*⟩ **}**

⟨*pattern-body*⟩ ::= ⟨*exec-clause*⟩ {**,** ⟨*exec-clause*⟩}

⟨*exec-clause*⟩ ::= **exec** action-name [⟨*when-clause*⟩]

⟨*when-clause*⟩ ::= **when (** expression **)**

Each pattern starts with a name and a pattern keyword split by a colon, where the keyword is used to differentiate from action in that there is no decided order. After the pattern keyword, a pattern-body surrounded by a pair of braces aligns several exec-clauses. Each exec-clause describes a predicate statement to evaluate and according action will be executed when the predicate is true. The when-clause is optional, and according action will be executed anyway if the when-clause is omitted. All clauses after the one without the when-clause will be ignored in that only the first action passing the predicate will be executed.

The predicates can be written in arbitrary expressions that can be evaluated in current program state of the target program, and a variable called self is created to reference the object being traversed. Predicates written in arbitrary JavaScript code provide strong expression power. It enables users to match the object in rich means. For example, the object can be matched by its type:

```
primitive_value: pattern {
  exec number_action when (typeof self === 'number'),
  exec string_action when (typeof self === 'string'),
  exec object_action when (typeof self === 'object'),
  ...
}
```

The object also can be matched by its class:

```
object: pattern {
  exec array_action when (self instanceof Array),
  exec tree_action when (self instanceof Tree),
  exec graph_action when (self instanceof Graph),
  ...
}
```

Furthermore, users can specify arbitrary constraints on the fields. The predicate to match an instance of Point class where the the field x is positive and the field y is non-zero:

```
point: pattern {
  exec point_action when (self instanceof Point &&
                          self.x > 0 &&
                          self.y != 0),
  ...
}
```

3.1.3 Actions

Along with a pattern, one must specify a series of actions that can be executed when the evaluation result of according predicate is true. One action is made up of create actions and next actions. Create actions are used to create visual nodes. Each action creates only one visual node represented with a set of attributes, which are name and value pairs describing the node. For example, a node could be customized by label and shape. The specific types of visual nodes and their attributes will be described in the next section. Next actions are used to indicate the objects to be traversed next. Next actions have two modes to support both object literals and arrays. To traverse arrays, we provide foreach statement. The grammar of actions is showed in EBNF as follows:

⟨*action*⟩ ::= action-name **: action {** ⟨*action-body*⟩ **}**

⟨*action-body*⟩ ::= ⟨*action-clause*⟩ {**,** ⟨*action-clause*⟩}

⟨*action-clause*⟩ ::= ⟨*create-clause*⟩ | ⟨*next-clause*⟩

⟨*create-clause*⟩ ::= **create** [environment-var **=**] node-name
(⟨*attribute-clause*⟩ {**,** ⟨*attribute-clause*⟩} **)**

⟨*attribute-clause*⟩ ::= attribute **=** expression

⟨*next-clause*⟩ ::= ⟨*object-clause*⟩ | ⟨*foreach-clause*⟩

⟨*object-clause*⟩ ::= **next** expression **(** ⟨*environment-clause*⟩
{**,** ⟨*environment-clause*⟩} **)**

⟨*foreach-clause*⟩ ::= **foreach_next (** ⟨*environment-clause*⟩
{**,** ⟨*environment-clause*⟩} **)**

⟨*environment-clause*⟩ ::= environment-var **=** expression

Like predicates described in patterns, the values of attributes and environment variables can be either arbitrary expressions written in JavaScript or other visual nodes created previously. For example, a binary tree can be visualized with pretty simple visualjs code. The data structure of the binary tree and the visualjs code is shown as follows:

```
{
  "value": <number>,
  "left" : <tree node>,
  "right": <tree node>
}

tree: action {
  create node=tree_node(label = self.value),
  create tree_edge(from = parent, to = node),
  next self.left(parent = node),
  next self.right(parent = node)
}
```

Besides those data structures defined in object literals, we could also visualize an array of integer elements in a bar chart. The visualjs code is shown as follows:

```
array: action {
  foreach_next()
}

element: action {
  create bar(value = self)
}
```

Two actions are defined here. The first one is for traversing the array, and the other one is for visualizing a separate element.

3.1.4 Environments

When traversing an object, it is often useful to pass along state information from one object to its descendant objects. Therefore, we introduced environments. Environments keep a set of variables storing visual nodes or primitive values. Environments can be created in next actions and passed along to subsequently traversed objects.

3.1.5 Layouts

Once a set of visual nodes is created from a traversal of an object. It will be sent to the client and rendered by a particular layout manager of provided by D3.js. The attributes like label and color will be displayed.

Currently, the system only contains layout managers for trees and bar charts, with more are to be added. Each of these layout algorithms use default settings of D3.js, so we do not elaborate the layout algorithms in the paper.

3.2 From An Object to Visual Shapes

To generate valid data structure visualization and animation, we must at first prove the consistency between the program and visualization. A program is composed of a series of instructions which are executed orderly. Under the control of debugger, program is being paused until step requests come. Every time the program pauses, it represents a new program state.

What need to be proved is that the initial program state is being correctly visualized and every time the new program state is generated, the animation will respond to it correctly. However, program state contains too much information. A computer program stores data in variables, which represent storage locations in the computer's memory. Program state contains all contents of these memory locations. In contrast, users always have limited interest and perception at a time. We intend to help users understand the program from any angel he is interested in. Instead of visualizing the whole program state like Heapviz [6], this system always focuses on visualizing one object but allows switching targets at any time.

3.2.1 Variables versus Objects

In JavaScript, a variable is a storage location and an associated symbolic name which contains a value. JavaScript: The Definitive Guide [11] introduces about data types in JavaScript. JavaScript allows you to work with

three primitive data types: numbers, strings of text, and boolean truth values. JavaScript also defines two trivial data types, null and undefined, each of which defines only a single value.

In addition to these primitive types, JavaScript also supports a composite data type known as object. Just like other object-oriented programming languages, an object in JavaScript is composed of a collection of values with either primitive values or objects. An object can represent an unordered collection of named values or an ordered collection of numbered values. The latter case is called an array. Although array is also an object, it behaves quite differently and have to be considered specially throughout the paper.

JavaScript also defines a few other specialized kinds of objects. **Function** is a subprogram that can be called externally or internally in case of recursion. **Date** creates a object that represents a single moment in time. **RegExp** creates a object that represents a regular expression for matching text with a pattern. **Error** creates a object that represent syntax or runtime errors that can occur in a JavaScript program. Because this research focuses on user-customized data structures, these four types of objects have their own specific data structures hence will not be considered any specially.

We selected object as the target to visualize. This seems trivially different from previous research like the famous data structure visualization system, jGRASP [9], who uses variable as the target of visualization. Although there is no problem using variable in jGRASP because it is static visualization without any animation. Our research introduces animation hence the problem has tremendously changed. The fundamental difference is whether there exists substantive relationship between two consecutive states. Animation is visualizing the changes between two consecutive states, so it have to proceed on the former state. That is why variable can not be used here as the target of visualization in that even the variables with the same name may refer to different objects or go out of scope after stepping through the program. Both situations may lead to meaningless animation because original mapping relationship can not be adapted for new object with different structure.

However, variable is still used as the entry to start watching certain object. Modern debugger offers many sophisticated features such as stepping through the program, stopping at some event by means of breakpoint and examining the current state such as tracking the values of variables and stack trace. We can find that typical debugging prefer examining variable values to exploring snapshot of the heap. Therefore, we choose variable as the entry and the object referred by the variable will drive the following animation.

Figure 3.1 shows the system interface. The left half is the visualization view.

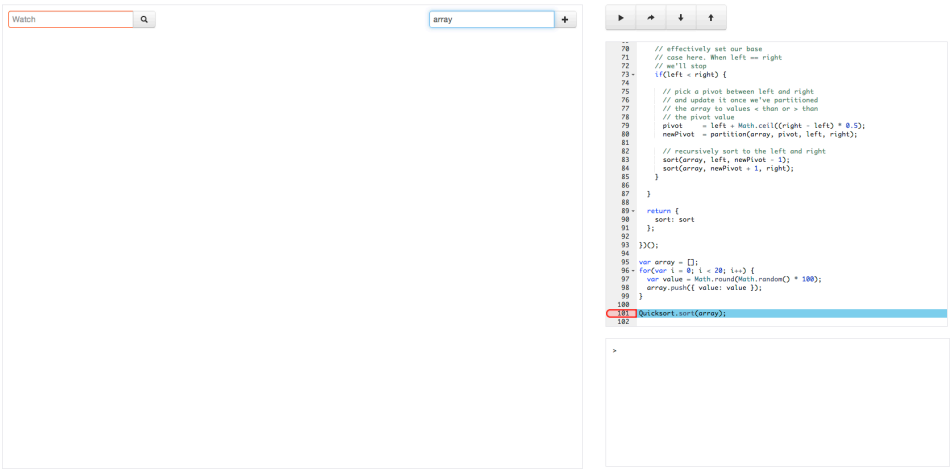

Figure 3.1: Select variable named array to visualize

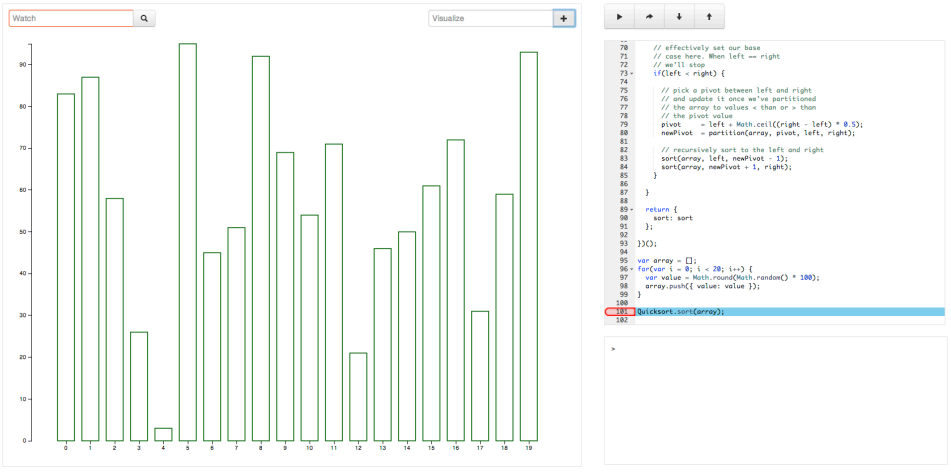

Figure 3.2: Variable named array has been visualized

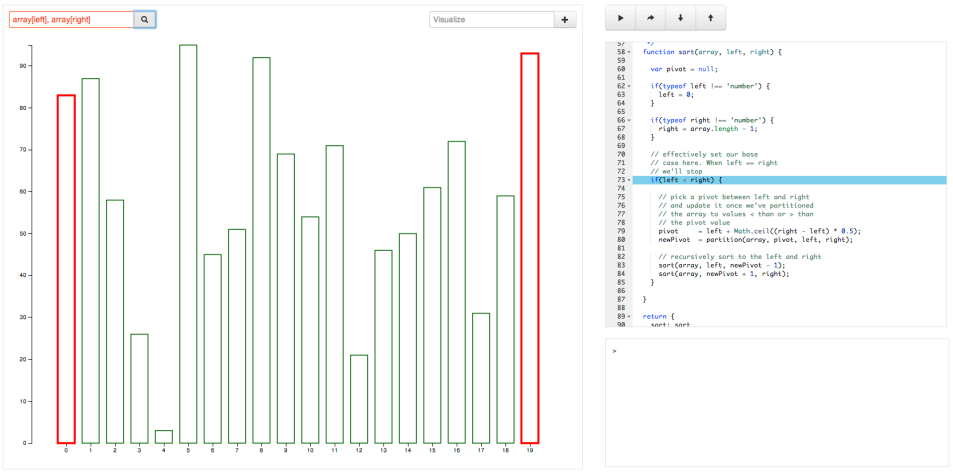

Figure 3.3: Context visualization

Users can select some object to visualize by means of inputting its variable name and click the button with add icon on the top right corner and then you can see that object has been visualized as a bar chart in Figure 3.2.

3.2.2 User Interest

As Niklaus Wirth said in his book [22], Algorithms + Data Structures = Programs, which means that we have to understand algorithms and data structures before understanding programs. Although our research is based on data structure visualization, we also intend to help users understand algorithms as far as possible. Although the attention has to be focused on one object, we still can help users understand algorithms from two aspects:

1. **Animation** displays how algorithms contribute to the object modification
2. **Highlighted shapes** represent the objects that users are interested in

In Figure 3.3, we can see that two bars are highlighted with red border against Figure 3.2. Users can input arbitrary number of variables split by commas in the top left corner of the visualization view. Any object referred by any inputted variable and possessing the same handle with any visual object will be highlighted with red border. Unlike that the target object being visualized is locked, these variables will be evaluated each step because they stand for how current environment relates with the target object.

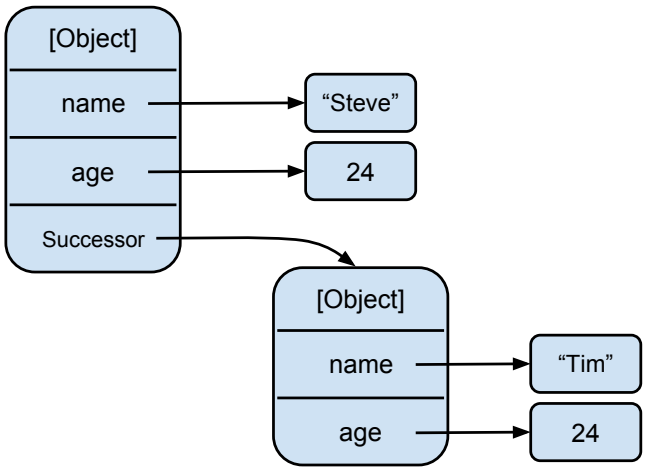

Figure 3.4: JavaScript object graph

3.2.3 Mapping Mechanism

Object graph details the relationships between objects. The object graph of the root object is defined as $G = (V, E)$. Here, the vertex set V represents all objects referred by the root object either directly or indirectly, and the edge set E represents the reference relationships between those objects. Figure 3.4 shows an example of JavaScript object graph.

Traversing the object graph according to the next actions, we will get a subgraph of it, $G_s = (V_s, E_s)$. The whole object graph probably contains much information that users are not interested in, hence we extract a subgraph from the whole object graph. All of the objects in the subgraph serve the visualization work, so we call them visual objects here.

After, each visual object is responsible to create its own visual shapes. We called them visual nodes here. Each data structure defines its own visual nodes. In this paper, we have implemented two data structures, bar chart and tree.

Bar chart only has one type of visual node. All visual nodes are connected into one list. The bars will be displayed in the same order of the list elements.

The data structure of the visual node is defined as follows:

```
{
  "value": <number>,
  "id"   : <string>
}
```

Property value is displayed via the bar height. Property id is a GUID representing the identification of visual node. If the visual nodes are created by objects, their id is set as the object handle. As for the visual nodes created by primitive type values, their id is set as their index in the list.

Tree has two types of visual nodes, node and edge. Their data structure are defined as follows:

```
{
  "label": <string>,
  "id"   : <string>,
  "type" : "node"
}
```

Node type visual node is displayed as a rectangle and property label can be seen inside. Like the visual node defined above, property id is also a GUID representing the identification of the visual node. For the visual nodes created by objects, their id is set as the object handles. For the visual nodes created by primitive type values, they must be leaf nodes in that there is no objects to traverse further. Their id is made up of three parts: the id of the adjacent node, the value of property label, and their index in the children node list. When the sibling nodes are also created by primitive type values, the latter two parts can be used to distinguish the visual nodes from their sibling nodes.

```
{
  "from": <node type visual node>,
  "to"  : <node type visual node>,
  "id"  : <string>,
  "type": "edge"
}
```

Edge type visual node is displayed as an undirected line linking two node type visual nodes, namely property from and property to. Property id has just the same meaning defined before. It is set as the id of the visual node expressed by property to in that the child node is always unique in a tree.

The visual nodes of the tree data structure can be defined as a graph, $G' = (V', E')$. The vertex set represents all visual nodes of tree node. V_s can be mapped to V' by a bijective function $f : V_s \to V'$. The edge set represents all visual nodes of tree edge. E_s also can be mapped to E' by a bijective function $f : E_s \to E'$.

Figure 3.5 shows the whole mapping process starting from the root object.

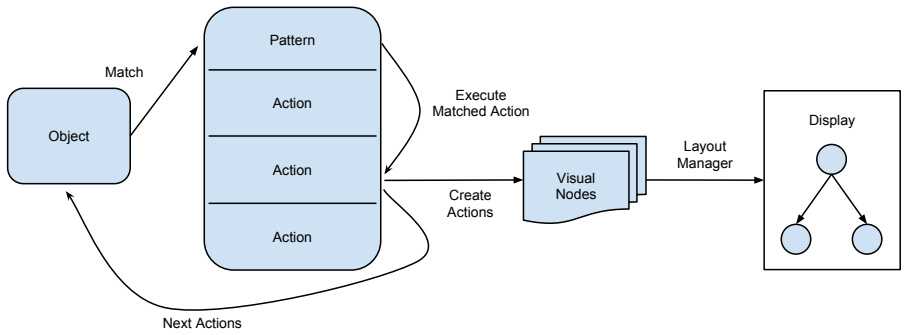

Figure 3.5: Mapping process

3.2.4 Animate Data Structures

After establishing the mapping relationship, we will be able to generate the animation. Animation is driven by data modifications, which is contributed by JavaScript program execution. Therefore, what we need to prove is the consistency between data modifications and animation. Figure 3.6 shows the whole process of animation generation.

The first step of proof is to enumerate all of the situations of data modifications and animation, and then data modifications can be translated to animation. Here, data is the visual nodes that have been explained above.

Data structure bar chart has following situations of data modifications:

1. A visual node with a new id was added to the list.
2. The index of some visual node with an existed id was updated.
3. A visual node with an existed id was removed from the list.
4. The property value of some visual node with an existed id was updated.

They can be translated to following animations on the bar chart representation:

1. The former three situations will contribute to recalculation of the layout. Those bars with either new size or position will be resized and translated smoothly.
2. The last situation will contribute to the update of bar height.

Data structure tree has following situations of data modifications:

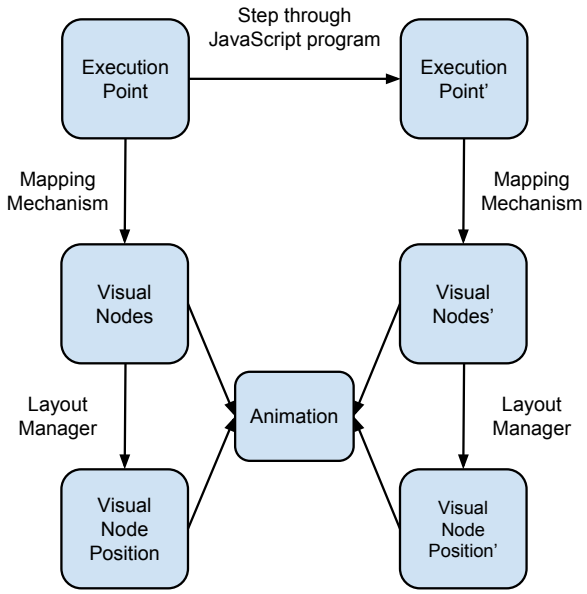

Figure 3.6: Animation generation process

1. A node type visual node with a new id was added.
2. A node type visual node with an existed id was removed.
3. Either the property from or to of some edge type visual node with an existed id was updated.
4. The property label of some node type visual node with an existed id was updated.

They can be translated to following animations on the tree representation:

1. The former three situations will contribute to recalculation of the layout. Those nodes and edges with new positions will be translated smoothly.
2. The last situation will contribute to the update of label text.

Chapter 4

System Implementation

4.1 V8 Debugger Protocol

V8 is able to debug the JavaScript code running in it. The debugger related API can be used in two ways, a function based API using JavaScript objects and a message based API using JSON based protocol. The function based API is for in-process usage, while the message based API is for out-process usage. This system is implemented with message based API. The protocol packet is defined in JSON format and can to be converted to string [5].

4.1.1 Protocol Packet Format

All packets have two basic elements called seq and type. The seq field holds the consecutive assigned sequence number of the packet. And type field is a string value representing the packet is request, response or event. Each request will receive a response with the same request seq number as long as the connection still works. And additional events will be generated on account of particular requests or system errors. Each packet has the following structure.

```
{
  "seq" : <number>,
  "type": <type>,
  ...
}
```

A request packet has the following structure.

```
{
  "seq"       : <number>,
  "type"      : "request",
  "command"   : <command>,
```

```
  "arguments": { ... }
}
```

A response packet has the following structure. If command fails, the success field will be set as false and message field will contain an error message.

```
{
  "seq"        : <number>,
  "type"       : "response",
  "request_seq": <number>,
  "command"    : <command>,
  "body"       : { ... },
  "running"    : <is the VM running after sending the message>,
  "success"    : <boolean indicating success>,
  "message"    : <error message>
}
```

An event packet has the following structure.

```
{
  "seq"  : <number>,
  "type" : "event",
  "event": <event name>,
  "body" : ...
}
```

4.1.2 V8 Debugger Protocol Features

V8 debugger has various commands and events providing detailed runtime information. However, this research focus on the data structure visualization. Only following features are used in order to implement basic debugger features.

- **Request** continue
- **Request** evaluate
- **Request** lookup
- **Request** source
- **Request** setbreakpoint
- **Request** clearbreakpoint
- **Event** break
- **Event** exception

Request "continue" makes V8 start running or stepping forward, including stepping in, stepping over, and stepping out. Although step count can be indicated in the arguments, we always set it as 1.

Request "evaluate" is used to evaluate a expression. However, if the result is object type that contains other fields, all fields will be represented as their object handle. Hence we have to use request "lookup" to lookup objects based on their handle. As a result, we can get the deep copy of any object by recursively using request "lookup".

Request "source" is used to retrieve source code for a frame. Frame and code range have to be indicated in the arguments. Note here that each script file running on node.js is wrapped within a wrapper function. Hence we have to remove the header and tail before showing it to users.

Request "setbreakpoint" is used to add breakpoint. Target file/function and line number are essential here. Request "clearbreakpoint" is used to remove breakpoint set by request "setbreakpoint". Breakpoint number which can be received from request "setbreakpoint" has to be indicated in the arguments. There also exists other kinds of requests like request "backtrace" which is used to require stacktrace information, request "frame" which is used to require frame information and so on.

4.1.3 Response Object Serialization

As discussed in 4.1.2, request "evaluate" and "lookup" may contain objects as part of the body. All objects are assigned with an ID called handle. Object identity[19] is that property of an object which distinguishes each object from all others. Although the handle can be used to identify objects here, it has a certain lifetime after which it will no longer refer to the same object. The lifetime of handles are recycled for each debug event.

For objects serialized they all contains two basic elements, handle and type. Each object has following the structure.

```
{
  "handle": <number>,
  "type"  : <"undefined", "null", "boolean", "number",
             "string", "object", "function">,
  ...
}
```

For primitive JavaScript types, the value is part of the result.

- 0 →

  ```
  {
    "handle": <number>,
    "type"  : "number",
    "value" : 0
  }
  ```

- "hello" →

```
{
  "handle": <number>,
  "type"  : "string",
  "value" : "hello"
}
```

- true →

```
{
  "handle": <number>,
  "type"  : "boolean",
  "value" : true
}
```

- null →

```
{
  "handle": <number>,
  "type"  : "null",
}
```

- undefined →

```
{
  "handle": <number>,
  "type"  : "undefined",
}
```

An object is encoded with additional information.

{a:1,b:2} →

```
{
  "className"           : "Object"
  "constructorFunction": { "ref": <number> },
  "handle"              : <number>,
  "properties"          : [{ "ref": <number> }, ...],
  "protoObject"         : { "ref": <number> },
  "prototypeObject"     : { "ref": <number> },
  "text"                : "#<Object>",
  "type"                : "object"
}
```

An function is encoded as an object with additional information in the properties name, inferredName, source and script.

function(){} →

```
{
  "handle"              : <number>,
  "type"                : "function",
  "className"           : "Function",
  "constructorFunction": { "ref": <number> },
```

```
  "protoObject"         : { "ref": <number> },
  "prototypeObject"     : { "ref": <number> },
  "name"                : "",
  "inferredName"        : "",
  "source"              : "function(){}",
  "script"              : { "ref": <number> },
  "scriptId"            : <number>,
  "position"            : <number>,
  "line"                : <number>,
  "column"              : <number>,
  "properties"          : [{
                              "name": <string>,
                              "ref" : <number>
                           }, ...]
}
```

4.2 System Architecture

This is a full-stack JavaScript system which consists of three components.

1. The debuggee node.js program is running on V8.
2. Server is also running on node.js.
3. Client is running on browser.

Server side is responsible for starting running debuggee program along with V8 debugger and communicate with it using V8 debugger protocol. It responds to the client and require according information from V8 debugger. When response arrives, this component is also responsible for informing client to update along with required information. Because the communication with V8 debugger is an asynchronized process whereas the system logic is basically synchronized, this component must be able to handle it carefully. Client side provides IDE-like debugging experience with an embedded editor to show the source code of the debuggee program, breakpoint management and series of stepping buttons. It also be responsible for visualization work. Figure **??** shows the system architecture.

4.2.1 Implementation Technique

Protocol.js ensures data integrity in that some responses may be separated into several chunks. Client.js is provides basic features by encapsulating the communication with V8 debugger. Animator module generates and updates graph for target object with the given script. It is feasible to build different interfaces, like command interface or GUI interface.

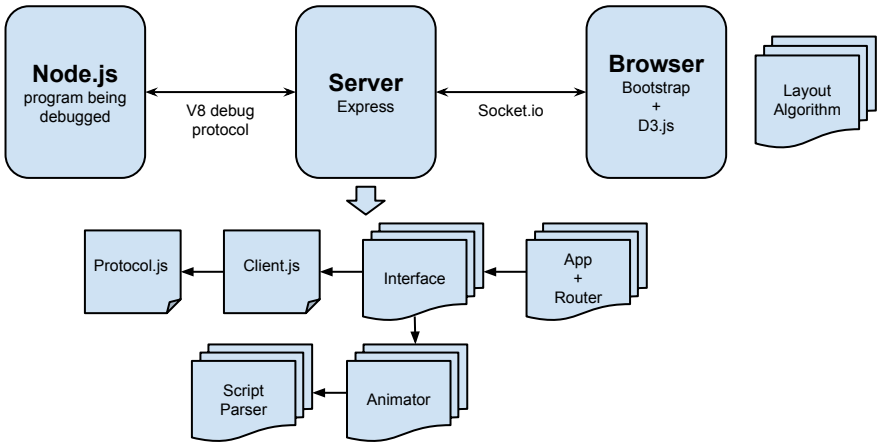

Figure 4.1: System architecture

On the client side, we used bootstrap for faster and easier web GUI development. Visualization related work is finished by D3.js which is an open-source project about visualizing. D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG and CSS. D3 's emphasizes on combining powerful visualization components with a data-driven approach to DOM manipulation.

We also used other open-source libraries and tools like underscore, async.js, jquery, require.js, buckets, grunt, jasmine to help the development work on both sides.

4.2.2 Asynchronized Communication Mechanism

Another mentionable implementation technique used here is the asynchronized communication mechanism. Most I/O-related API containing TCP/IP communication provided by node.js is in asynchronized way. On the other hand, all requests sent from client via websocket is also in asynchronized way. As a result, it is very complicated to keep program running correctly because the overall control flow across three components is based on synchronization logic. Firstly, blocking message queue is used to handle the requests

from client. All requests are stored in the queue and will be handled one by one. On the server side, Async.js is used to help manage asynchronized code. Although node.js is famous for its speed and single-thread model, its coding style is difficult to maintain on account of endless nested callback. Async.js is a tool to help alleviate this problem.

4.3 VisualJS Parser

We use PEG.js here to parse VisualJS code. PEG.js is a simple parser generator based on parsing expression grammars [12] for JavaScript that produces fast parsers with excellent error reporting. It is used to process complex data or computer languages and build transformers, interpreters, compilers and other tools.

4.3.1 Using the Parser

PEG.js generates parser from a grammar that describes expected input and can specify what the parser returns using semantic actions on matched parts of the input. Generated parser itself is a JavaScript object with a simple API.

Generated parser can be used by calling the parse method with an input string as a parameter. The method called parse will return a parse result or throw an exception if the target is invalid. The exception contains location information and other detailed error messages.

4.3.2 Grammar Syntax and Semantics

The grammar syntax is similar to JavaScript in that it is not line-oriented and ignores whitespace between tokens.

An example of parsing simple arithmetic expressions like 6/(1+2) is shown as follows. A parser generated from the grammar is able to calculate the expression results.

```
start
  = additive

additive
  = left:multiplicative "+" right:additive {
    return left + right;
  }
  / multiplicative
```

```
multiplicative
  = left:primary "*" right:multiplicative {
    return left * right;
  }
  / primary

primary
  = "(" additive:additive ")" {
    return additive;
  }
  / integer

integer
  = digits:[0-9]+ {
    return parseInt(digits.join(""), 10);
  }
```

4.3.3 Parsing Expression Types

There are a series of parsing types, and some of them contain subexpressions forming a recursive structure [4]:

- **"literal"**

  Match literal string and return it.

- **.**

  Match an arbitrary character and return it as a string.

- **[characters]**

  Match one character from a set and return it as a string. We can set a range on characters. For example, [a-z] means all lowercase letters. Preceding the characters with ^ inverts the matched set. For example, [^a-z] means all character but lowercase letters.

- **(expression)**

  Match a subexpression and return its match result.

- **expression ***

  Match zero or more repetitions of the expression and return the match results in an array. The matching is greedy. The parser will match the expression as many times as possible.

- **expression +**

  Match one or more repetitions of the expression and return the match results in an array. The matching is greedy. The parser will match

the expression as many times as possible.

- **expression ?**

  Try to match the expression. If the match succeeds, the match result will be returned, otherwise null will be returned.

- & **expression**

  Try to match the expression. If the match succeeds, just return undefined and do not advance the parser position, otherwise consider the match failed.

- **! expression**

  Try to match the expression. If the match does not succeed, just return undefined and do not advance the parser position, otherwise consider the match failed.

- **$ expression**

  Try to match the expression. If the match succeeds, return the matched string instead of the match result.

- **label : expression**

  Match the expression and remember its match result under given label.

- **expression1 expression2 ... expressionn**

  Match a sequence of expressions and return their match results in an array.

- **expression { action }**

  Match the expression. If the match is successful, run the action, otherwise consider the match failed. The action is a piece of JavaScript code that is executed as if it was inside a function. It gets the match results of labeled expressions in preceding expression as its arguments. The action should return some JavaScript value using the return statement. This value is considered match result of the preceding expression.

- **expression1 / expression2 / ... / expressionn**

  Try to match the first expression, if it does not succeed, try the second one, etc. Return the match result of the first successfully matched expression. If no expression matches, consider the match failed.

Chapter 5

Examples

To best explain how VisualJS converts an object to visual shapes, some typical examples will be talked below.

5.1 Quick Sort

The first example considers a quick sort implementation in JavaScript. The data to be sorted is stored in an array, and the data structure of the element is defined as follows:

```
{
  value: <number>
}
```

The data structure definition is enough to write VisualJS code. Because VisualJS is interpreted dynamically, users can modify the VisualJS code at any time. The VisualJS code used here is as follows:

```
array: pattern {
  exec array when (self instanceof Array),
  exec element when (typeof self === 'object')
}

array: action {
  foreach_next()
}

element: action {
  create bar(value = self.value)
}
```

5.2 AVL Tree

The second example is about AVL tree.

```
operator: pattern {
  exec avltree_node
}

avltree_node: action {
  create node=tree_node(label = self.node[self.attr]),
  create tree_edge(from = parent, to = node),
  next self.left(parent = node),
  next self.right(parent = node)
}
```

5.3 Math Expression Tree

The third example is about math expression tree.

```
operator: pattern {
  exec plus when (self.op === 0),
  exec minus when (self.op === 1),
  exec times when (self.op === 2),
  exec divide when (self.op === 3),
  exec value when (typeof self.value === 'number')
}

plus: action {
  create node=tree_node(label = '+'),
  create tree_edge(from = parent, to = node),
  next self.left(parent = node),
  next self.right(parent = node)
}

minus: action {
  create node=tree_node(label = '-'),
  create tree_edge(from = parent, to = node),
  next self.left(parent = node),
  next self.right(parent = node)
}

times: action {
  create node=tree_node(label = '*'),
  create tree_edge(from = parent, to = node),
  next self.left(parent = node),
  next self.right(parent = node)
}

divide: action {
  create node=tree_node(label = '/'),
  create tree_edge(from = parent, to = node),
```

```
  next self.left(parent = node),
  next self.right(parent = node)
}

value: action {
  create node=tree_node(label = self.value),
  create tree_edge(from = parent, to = node)
}
```

Chapter 6

Summary

6.1 Future Work

Bibliography

[1] Chrome developer tools homepage. `https://developer.chrome.com/devtools/index`.

[2] Ddd - data display debugger documention. `http://www.gnu.org/software/ddd/manual/`.

[3] Gdb documentation. `https://developer.chrome.com/devtools/index`.

[4] Peg.js documentation. `http://pegjs.majda.cz/documentation`.

[5] V8 debugger protocol. `https://code.google.com/p/v8/wiki/DebuggerProtocol`.

[6] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 53–62, New York, NY, USA, 2010. ACM.

[7] L.A Barowski and II Cross, J.H. Extraction and use of class dependency information for java. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 309–315, 2002.

[8] II Cross, J.H., T.D. Hendrix, and L.A Barowski. Using the debugger as an integral part of teaching cs1. In *Frontiers in Education, 2002. FIE 2002. 32nd Annual*, volume 2, pages F1G–1–F1G–6 vol.2, Nov 2002.

[9] James H. Cross, II, T. Dean Hendrix, Jhilmil Jain, and Larry A. Barowski. Dynamic object viewers for data structures. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 4–8, New York, NY, USA, 2007. ACM.

[10] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[11] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 1998.

[12] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, January 2004.

[13] Paul Gestwicki and Bharat Jayaraman. Methodology and architecture of jive. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 95–104, New York, NY, USA, 2005. ACM.

[14] Paul V. Gestwicki and Bharat Jayaraman. Jive: Java interactive visualization environment. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 226–228, New York, NY, USA, 2004. ACM.

[15] Philip J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[16] Juha Helminen and Lauri Malmi. Jype - a program visualization and programming exercise tool for python. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 153–162, New York, NY, USA, 2010. ACM.

[17] T. Dean Hendrix, James H. Cross, II, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. *SIGCSE Bull.*, 36(1):387–391, March 2004.

[18] Ville Karavirta and Clifford A. Shaffer. Jsav: The javascript algorithm visualization library. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 159–164, New York, NY, USA, 2013. ACM.

[19] Setrag N. Khoshafian and George P. Copeland. Object identity. *SIGPLAN Not.*, 21(11):406–416, June 1986.

[20] J.L. Korn and A.W. Appel. Traversal-based visualization of data structures. In *Information Visualization, 1998. Proceedings. IEEE Symposium on*, pages 11–18, Oct 1998.

[21] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223, December 2007.

[22] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.