1 Introduction

The typical process of using debugger in IDE is to step through the program with examining program state (values of variables and the call stack) to understand the program and detect bugs.
In visualJS, our goal is to animate an arbitrary data structure implementation without breaking the normal debugging experience. To achieve this goal, we won't insert anything into source code and permit users to select and modified visualization method and target object dynamically. Most of current tools can't meet these two requirements. Consequently, their usage scenarios are limited.

2 Animation Mechanism

To generate valid data structure animation, we must prove the consistency between the visualization and program. 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 we need to prove is that every time the new program state has been generated, the animation will respond to it correctly.

2.1 Visual Nodes

Object graph details the relationships between objects. We can get one object graph $G = (V, E)$ for a given object at certain program state. The vertex set V_P is defined by objects which refer to that given object directly or indirectly, and the edge set E is defined by reference relationships among those objects.
By iterating the object graph using next actions, we will get a subgraph of it, $G_p = (V_p, E_p)$. Visual nodes are defined based on this graph.
Visual nodes including two types, node and edge, are created by create actions. They are also a graph, $G_v = (V_v, E_v)$. The vertex set is defined by visual nodes of node type. They are related to V_p. The relationships can be defined as a non-injective and surjective function $f : V_p \to V_v$ (***TODO***: illustration). The edge set is defined by visual nodes of edge type.

2.2 Visualization State

3 Example Explanation

```
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 === 'number')
}
```