Code Clones Detection Based on Semantic Analysis for JavaScript Language*

Sevak, Sargsyan
ISP RAS
Moscow, Russia
e-mail: sevaksargsyan@ispras.ru

Shamil, Kurmangaleev
ISP RAS
Moscow, Russia
e-mail: kursh@ispras.ru

Vahagn, Vardanyan
ISP RAS
Yerevan, Armenia
e-mail: vaag@ispras.ru

Vachagan, Zakaryan
ISP RAS
Yerevan, Armenia
e-mail: zakaryan@ispras.ru

ABSTRACT
Existed methods of code clones detection for JavaScript programs are based on textual, lexical or syntactic analysis. These methods have relatively low accuracy and cannot detect strongly modified fragments of code. The article describes a new method of code clone detection for JavaScript programming language based on semantic analysis of the program. Due to using of Program Dependence Graphs (PDG) the method detects strongly modified fragments of code as well as exact clones. It has high accuracy and is scalable for analysis of million lines of source code. Comparison results with CloneDR have shown that the proposed method detects about ten times more clones with recall higher than sixty percentages. Manual analysis of detected clones has shown that the rate of false positive for new method is lower than five percentages.
The method is implemented as part of V8 JavaScript compiler. It generates PDG graphs for JavaScript functions based on V8's intermediate representation named Hydrogen. For generated PDGs a special tool is applied to detect code clones. The tool detects maximal isomorphic subgraphs, which are considered as clones. Set of scripts are developed for parallel run of the tool in multiprocessor systems and analyzing detected clones.

Keywords
Code clone, PDG, JavaScript, V8

1. INTRODUCTION
JavaScript language has become very popular due to its minimal verbosity, code maintainability, and ease of rapid prototyping. JavaScript is now used not only for executing small scripts in web browsers, but also as the main language for developing applications on some operating systems for mobile and media devices, such as Tizen [1] or FirefoxOS [2]. During the development of large software systems copy-paste activities by the software developers become usual. Reusing code can lead to many semantic errors. For example, software developer can forget to rename some variables or functions after copy-paste. The software, which has many clones, probably will have many mistakes and low quality. According to different studies [3, 4] up to 20 percent of source code can be clone in software. Clone detection tools are widely used during software maintaining. It allows to avoid mistakes and improves software quality.
The goal of this paper is to introduce a code clone detection method for JavaScript programming language, based on semantic analysis. It consists of two basic stages. The first part transforms V8's Hydrogen representation of JavaScript source to PDG. PDGs are constructed during execution of Crankshaft (see section 2.1) and serialized into files. The second part is a separate tool [5] for analyzing the stored PDGs to detect code clones. It detects maximal isomorphic subgraphs as code clones. Introduced method of clone detection has a number of applications for JavaScript projects:

1. Automatic refactoring.
2. Code size optimizations. Repeated fragments of code can be replaced by call of one function.
3. Semantic errors detection. If one fragment of code contains a semantic error all clones of this fragment will have the same error with high probability.

Three scripts are provided for parallel run of the tool [5] in multiprocessor systems and analyzing detected clones.

2. BACKGROUND
2.1. V8 JIT compiler
V8 is Just-in-time compiler for JavaScript language. It has two separate compilers for source code compilation into machine code (Fig. 1). The first compiler is Full-Codegen, which compiles source code directly into machine code in order to produce code quickly. The second compiler is called Crankshaft, which is slower and produces an optimized machine code. At first V8 parses source code into abstract syntax tree (AST) and uses Full-Codegen to produce the machine code quickly. During the execution of code generated by Full-Codegen profile information for the program is collected, such as type information, inline caches, etc. At the same time runtime profiler samples JavaScript code in order to determine hot (frequently executed) functions.

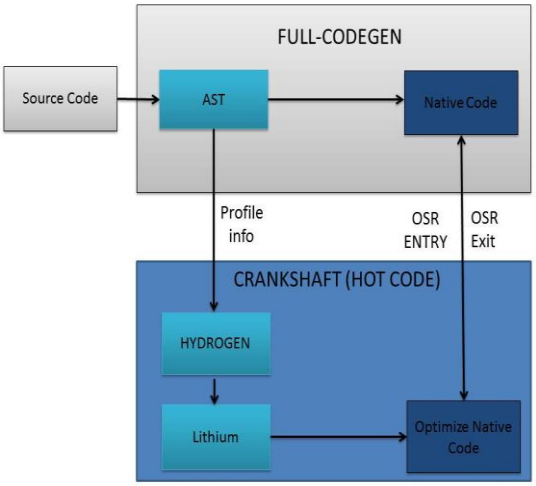

Figure 1. V8 JIT Architecture

Hot functions are recompiled by the Crankshaft compiler. Crankshaft translates AST code into control flow graph (CFG) with SSA-like representation called Hydrogen. Collected type and other runtime information allow Crankshaft to optimize the functions speculatively, under the

* The paper is supported by RFBR grant 15-07-07541

assumption that certain properties of the functions will not change during the next run. Hydrogen representation is used to implement many well-known optimizations, such as dead code elimination, constant propagation, common subexpression elimination, bounds redundant check elimination, loop invariant code motion, etc. During execution, if the optimized code encounters a case that it cannot handle (for example, when type of value of the variable does not match profile information), it bails to the code generated by Full-Codegen. This transition is called an on-stack replacement. After all optimizations are performed on Hydrogen graph it is transformed to low-level, machine-dependent intermediate representation called Lithium. This representation is closer to three-address code, with labels and "goto" instructions. Each Lithium instruction has its output, input and temporary operands. Register allocation is performed in Lithium representation, and then a binary code is generated.

2.2. Clone types

Clone types are categorized in three basic groups [6]. The first group is identical code fragments except the variations in whitespaces (T1). The second group is identical code fragments except the variations in identifiers, literals, types, layout and comments (T2). The third group is copied fragments of code with further modifications.

2.3. Code clone detection approaches

Numbers of approaches [7, 8] were provided for code clones detection, but they have some restrictions. Textual approach [9] considers the source code of the program as a text and tries to find matched substrings as code clones. When all clones are detected, the clones which are located nearby can be combined into one. Methods based on lexical approach [10, 11] parse source code to sequence of tokens. Longest common subsequences of tokens are considered as code clones. These two approaches cannot detect clones of (T3) type. In case of syntactic approach [12, 13] AST is analyzed instead of source code. Code clones are matched subtrees of AST. Methods based on this approach are more effective for detecting clones of (T1) and (T2) types; (T3) types of clones are detected with low accuracy, because the added or deleted instructions strictly change the structure of AST. Methods based on semantic analysis [5, 14, 15] translate source code to PDG. Nodes of PDG are instructions of the program. Edges of PDG are dependences between the instructions. Isomorphic subgraphs of PDG are considered as code clones. Algorithms based on semantic analysis have high computational complexity, but able to detect all three types of clones with high accuracy. Metrics-based algorithms [16, 17, 18] compute a number of metrics for code fragments and compare them. This approach has low accuracy. For qualitative analysis of software systems, (T3) clones should be detected as well as others. The tool [5] which we use for code clone detection is semantic based. It allows detecting all three types of clones with high accuracy.

3. TRANSLATION FROM HYDROGEN TO PDG

3.1. Hydrogen

Hydrogen's structure is very similar to intermediate representation of LLVM [19]. Hot functions are parsed to abstract syntax tree (AST). Based on AST and profile information Hydrogen is constructed. It is represented as a control flow graph of basic blocks where each block contains a sequence of instructions in static single assignment (SSA) form. Each instruction has a list of operands and a list of uses. So Hydrogen is a data flow graph being layered on top of the control flow graph. Each Hydrogen instruction represents a fairly high-level operation, such as an arithmetic operation, a property load or store, a function call or a type check.

3.2. PDG

Program dependence graph is the most detailed representation for the program. It contains control and data flow information, information about variables aliasing. PDG is a directed graph where nodes are instructions, edges are dependences between the instructions. It contains two basic types' of edges. The first type is control edges which represent the control flow graph for the program. The second type is data dependences, which has three categories for pair of nodes:

1. True-dependence means that value written to the memory by the first instruction is read second;
2. Anti-dependence means that value written to the memory by the second instruction is read first.
3. Output-dependence means that two instructions are write to the same memory.

3.3. Translation

For every instruction of Hydrogen a new PDG node is constructed. Type of constructed node is determined based on type of the corresponding Hydrogen instruction. Every node has information about source code line from which it was constructed. Between two nodes of PDG a data edge is added if the corresponding instructions of Hydrogen have true-dependence, anti-dependence or output-dependence. Control dependences are added based on analysis of Hydrogen's basic blocks. Between two nodes of PDG a control edge is added if the instruction corresponding to the first node executed before the instruction corresponding to the second node. When Hydrogen is translated to PDG it is serialized to file. Serialization format is acceptable for the tool [5] used for clone detection.

4. ANALYZE SCRIPTS

Three scripts are developed for parallel run of the tool and analyzing detected code clones. The first script is responsible for parallel run of the tool in multiprocessor systems. The tool as input takes one or two lists of PDGs. For the single list input the tool makes pairwise comparison of all PDGs from this list. In case of two lists, the tool compares PDGs from different lists. The script splits the list of PDGs for the project to smaller lists. Number of split lists depends on the number of processors and memory size. For all single small lists and different pairs of these lists the instance of the tool is run. In case of p processors PDGs' list should be split at least to x smaller lists, where $x = \frac{-1+\sqrt{1+8p}}{2}$. Otherwise the resources of target machine will not be fully used. If the memory of target machine is not enough for analyzing of pair of sublists then the number of sublist should be increased.

The second script is responsible for visualization of detected clones. It allows showing source code and corresponding PDG graphs for detected clone. The script supports HTML and EXEL output formats for detected clones.

The third script analyzes files of detected clones. It allows tracking history of cloned fragments of code for the files. Tree of files is constructed based on cloning history. User can interactively move through the tree of files and follow modifications of the cloned fragments of code.

5. EXPEREMENTAL RESULTS

5.1. Detected clones

Developed method was applied for three widely used JavaScript benchmarks. Target machine is Intel Core 2 Duo CPU E7400 with 8 GB of RAM. Minimal clone length is 10 lines of source code and similarity higher than 90%.

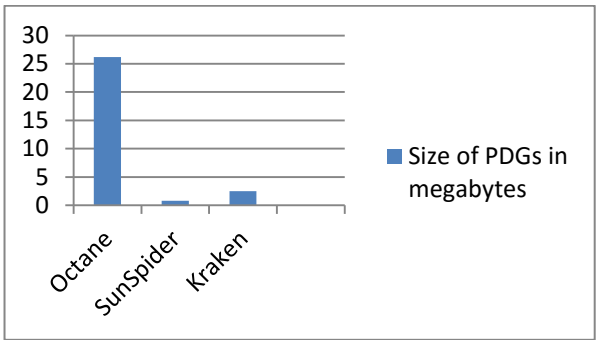

Figure 2. Size of PDGs.

Figure 2 shows sizes of generated PDGs for JavaScript benchmarks.

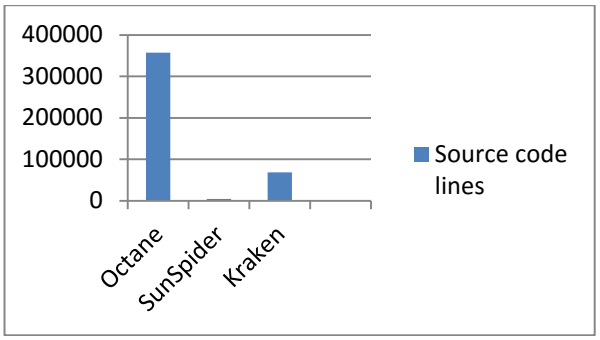

Figure 3. Source code lines.

Figure 3 shows lines of source code for analyzed projects.

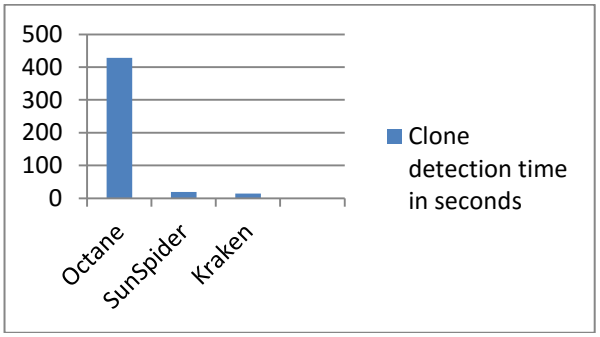

Figure 4. Clone detection time.

Figure 4 shows run time of the clone detection tool [5] for generated PDGs.

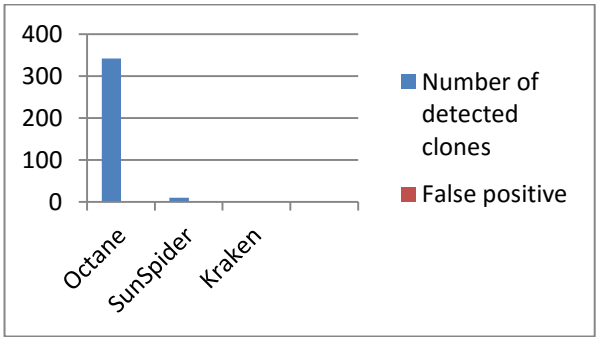

Figure 5. Number of detected clones.

Figure 5 shows the number of detected code clones and the rate of false positive. For example, for the Octane benchmark 342 code clones are detected and 5 of them are false positive. Clone detection time is 428 seconds. Octane has about 357.000 lines of source code written in JavaScript. SunSpider has 10 code clones, only one is false positive. For the Kraken tool does not detect any clone.

5.2. Comparison with CloneDR

Developed method was compared with the tool CloneDR [20], which is developed by Semantic Designs Company. The company provides different tools for software design and analyses. Minimal clone length is 10 lines of source code and similarity higher than 90%. Target test is octane benchmark. CloneDR has detected 35 clones. Our tool has detected 342 clones, 21 of them were common with CloneDR results.

6. FUTURE WORK

V8 is JIT compiler so functions are compiled if only they are called. It means that PDG graphs are generated only for the functions which were called during the execution time. If some fragments of JavaScript code are clones, but not executed, they will not be detected as clones. We are planning to make V8 generate Hydrogen graphs for all functions ether they are called during the execution time or not.

7. ACKNOWLEDGEMENT

The paper is supported by RFBR grant 15-07-07541.

REFERENCES

[1] Tizen platform website. http://tizen.org/

[2] Mozilla website. https://www.mozilla.org

[3] B. Baker, "On finding duplication and near-duplication in large software systems", *in: Proceedings of the 2nd Working Conference on Reverse Engineering*, WCRE 1995, pp. 86-95, 1995.

[4] C.K. Roy, J.R. Cordy, "An empirical study of function clones in open source software systems", *in: Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE 2008, pp. 81-90, 2008.

[5]. S. Sargsyan, S. Kurmnagaleev, A. Belevantsev, H. Aslanyan, A. Baloian, "Scalable code clone detection tool based on semantic analysis", *The Proceedings of ISP RAS*, pp. 39-49 2015.

[6]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., "Comparison and evaluation of clone detection tools", *Transactions on Software Engineering 33* pp. 577–591, 2007.

[7]. D. Rattana, R. Bhatiab and M. Singhc, "Software clone detection", *Information and Software Technology*, vol. 55, no. 7, pp. 1165-1199, 2013.

[8]. S. Gupta and P. C. Gupta, "Literature Survey of Clone Detection Techniques", *International Journal of Computer Applications*, vol. 99, no. 3, pp. 41-44, 2014.

[9]. S. Ducasse, M. Rieger, S. Demeyer, "A language independent approach for detecting duplicated code", *in: Proceedings of the 15th International Conference on Software Maintenance* (ICSM'99), Oxford, England, UK, pp. 109-119, 1999.

[10]. T. Kamiya, S. Kusumoto, K .Inoue, CCFinder: "A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, 2002.

[11]. R. Kaur, S. Singh, "Clone detection in software source code using operational similarity of statements", *ACM SIGSOFT Software Engineering Notes*, Volume 39 Issue 3, pp. 1-5, 2014.
[12] I. Baxter, A. Yahin, L. Moura, M. Anna, "Clone detection using abstract syntax trees", *in: Proceedings of the 14th IEEE International Conference on Software Maintenance*, IEEE Computer Society, pp. 368-377, 1998.
[13] L.Jiang, G.Misherghi, Z.Su, S.Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones", *in: Proceedings of the 29th International Conference on Software Engineering* (ICSE07), IEEE Computer Society, pp. 96-105, 2007.
[14] R.Komondoor, S.Horwitz, "Using slicing to identify duplication in source code", *in: Proceedings of the 8th International Symposium on Static Analysis*, pp. 40-56, 2001.
[15] J. Krinke, "Identifying similar code with program dependence graphs", *in: Proceedings of the 8th Working Conference on Reverse Engineering*, (WCRE 2001), pp.301-309, 2001.
[16] J. Mayrand, C. Leblanc, E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics", *in: Proceedings of the 12th International Conference on Software Maintenance* (ICSM96), Monterey, CA, USA, pp. 244-253, 1996.
[17] N. Davey, P. Barson, S. Field, R. Frank, "The development of a software clone detector", *International Journal of Applied Software Technology*, vol 1, no. 3/4, pp. 219-236, 1995.
[18]. S. S. Sargsyan, S. F. Kurmangaleev, A. V. Baloian, H. K. Aslanyan, "Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph", *Mathematical Problems of Computer Science 42*, pp. 54-62, 2014.
[19]. http://www.llvm.org
[20] http://www.semdesigns.com/products/clone/