Local optimization of JavaScript code

FÁBIO FARZAT
MÁRCIO BARROS
GUILHERME TRAVASSOS
Emphasis

- **Break things** instead of repairing them

- **Syntax tree** level manipulation instead of source code lines

- **Local search** instead of genetic algorithms
Why should one care about JavaScript?

- The world seems to be committed to JavaScript
- The first version of the programming language was developed in a couple of weeks to allow “non-programmers” handle the structure of a Web page in a browser
- It was designed to provide a better interaction model between the front- and the back-end of Web apps
- Now, JavaScript is used everywhere ...
JavaScript is everywhere ...

95% front-end
97% hybrid mobile
0.4% back-end

https://goo.gl/kWbgsU
https://ionicframework.com/survey/2017#trends
https://www.similartech.com/technologies/nodejs
The dangers of JavaScript

- At the same time, a future that depends so much on JavaScript is worrisome
- JavaScript shows peculiar behavior if a developer goes beyond the bounds of "normal" programming
The dangers of JavaScript

var ts
undefined
> ts
undefined
> ts * 1
NaN
> ts | 1
1
> 42.toFixed(2)
SyntaxError: Invalid or unexpected token
> 42.toFixed(2)
'42.00'

https://www.youtube.com/watch?v=2pL28CcEijU
https://www.destroyallsoftware.com/talks/wat
Objectives

The main objective of our research is to find variants of a target JavaScript program which are smaller and functionally-equivalent to the target program.
Objectives

The main objective of our research is to find variants of a target JavaScript program which are smaller and functionally-equivalent to the target program.

Reducing the size of the source code (minified version) will reduce load and processing times.
Objectives

The main objective of our research is to find variants of a target JavaScript program which are smaller and functionally-equivalent to the target program.

Equivalence as attested by the test suite of the target program, which acts as our (limited) oracle.
This is an ongoing work!

At the moment, we are interpreting the results collected from a second round of experiments.

But it all started with an opportunity ...
Opportunity strikes!

- The university installs a supercomputer and needs someone to test it!

- **Lobo Carneiro**
  - Cluster-based supercomputer
  - 252 processing nodes
  - Each node has 24 cores running HT
  - 16 Tb of RAM memory
  - 720 Tb of disk
Opportunity strikes!

- We examined the fitness landscape for JavaScript source code improvement

- We executed a genetic algorithm and a random search over 13 target programs
  - Mutation operator that removes nodes from the AST
  - 5,000 fitness evaluations/round, 60 rounds for each program

- At top usage, we occupied 2,880 cores and 500 Gb of RAM
The selected JavaScript programs

Heavily-used JavaScript libraries

>= 90% statement coverage

Distinct sizes, from small to large

Researchers had some experience

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th># Tests</th>
<th>% Cov</th>
<th>Usage</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browserify</td>
<td>757</td>
<td>570</td>
<td>99%</td>
<td>1.9</td>
<td>14.3.0</td>
</tr>
<tr>
<td>Executimer</td>
<td>195</td>
<td>37</td>
<td>91%</td>
<td>0.4</td>
<td>n/a</td>
</tr>
<tr>
<td>Express-ifttt</td>
<td>160</td>
<td>29</td>
<td>93%</td>
<td>0.2</td>
<td>n/a</td>
</tr>
<tr>
<td>Gulp-cccr</td>
<td>525</td>
<td>17</td>
<td>90%</td>
<td>0.1</td>
<td>n/a</td>
</tr>
<tr>
<td>jQuery</td>
<td>7,607</td>
<td>937</td>
<td>91%</td>
<td>4.8</td>
<td>3.2.0</td>
</tr>
<tr>
<td>Lodash</td>
<td>10,795</td>
<td>2,077</td>
<td>94%</td>
<td>33.8</td>
<td>3.10.1</td>
</tr>
<tr>
<td>Minimist</td>
<td>193</td>
<td>140</td>
<td>92%</td>
<td>25.8</td>
<td>1.2.0</td>
</tr>
<tr>
<td>Plivo-node</td>
<td>609</td>
<td>26</td>
<td>91%</td>
<td>10.6</td>
<td>n/a</td>
</tr>
<tr>
<td>Pug</td>
<td>400</td>
<td>240</td>
<td>98%</td>
<td>356</td>
<td>4.0.0</td>
</tr>
<tr>
<td>Tleaf</td>
<td>133</td>
<td>131</td>
<td>96%</td>
<td>0.2</td>
<td>n/a</td>
</tr>
<tr>
<td>Underscore</td>
<td>1,481</td>
<td>198</td>
<td>95%</td>
<td>8,710</td>
<td>1.8.3</td>
</tr>
<tr>
<td>UUID</td>
<td>209</td>
<td>21</td>
<td>91%</td>
<td>11,497</td>
<td>n/a</td>
</tr>
<tr>
<td>XML2JS</td>
<td>526</td>
<td>83</td>
<td>93%</td>
<td>3,783</td>
<td>0.4.16</td>
</tr>
</tbody>
</table>
Findings

- Surprisingly, random search outperformed genetic algorithms for all instances
  - GA failed to find improved versions in more than 50% of its runs for all programs
  - RD fails less frequently and found variants representing from 0.2% to 22% reduction!

- Patches are small and clustered in independent parts of the source code
  - The distance between patches is moderately and inversely correlated to program size
  - Patch size is strongly and inversely correlated to program size
  - The median is always smaller than the average for both measures (a few large values)
  - The best variants found by random search had many patches (37% rounds found 5+ patches)
Findings: an example

Listing 1. function getTimeFieldValues (original version of the source code)

```javascript
UUIDjs.getTimeFieldValues = function (time) {
    var ts = time - Date.UTC(1582, 9, 15);
    var hm = ts / 4294967296 + 10000 & 268435455;
    return {
        low: (ts & 268435455) * 10000 % 4294967296,
        mid: hm & 65535,
        hi: hm >>> 16,
        timestamp: ts
    };
}
```

Listing 2. function getTimeFieldValues (optimized version of the source code)

```javascript
UUIDjs.getTimeFieldValues = function () {
    var ts;
    var hm;
    return {
        low: (ts & 268435455) * 10000 % 4294967296,
        mid: hm & 65535
    };
}
```
Findings: patch distribution

These findings imply that basic genetic algorithms are not effective for JavaScript source code reduction because the chances that recombination merges independent mutations are very small.
Findings: patch distribution

UUIDjs.getTimeFieldValues = function(time) {
    var ts = time - Date.UTC(1582, 9, 15);
    var hm;
    return {
        low: (ts & 268435455) * 10000 % 4294967296,
        mid: hm & 65535,
        hi: hm >>> 16,
        timestamp: ts
    };
};

Individual 1 (subjected to one mutation)

UUIDjs.getTimeFieldValues = function(time) {
    var ts;
    var hm = ts / 4294967296 * 10000 & 268435455;
    return {
        low: (ts & 268435455) * 10000 % 4294967296,
        mid: hm & 65535,
        hi: hm >>> 16,
        timestamp: ts
    };
};

Individual 2 (subjected to a second mutation)

What are the chances of a one-point crossover that keeps both building blocks?
Findings: patch distribution

Rephrasing: what are the chances of selecting these cutting points?

They are inversely proportional to the square of the number of instructions in the target program.
So, what is the alternative?

- A systematic transversal of the search space (for instance, a **local search**) may find better results than random search
  - Local search behaves well if departing from a good solution (the human-written program)
  - Optimization is performed by removing nodes from the AST that do not contribute to the test cases
  - The **key challenge** is the size of the neighborhood for any given program
JavaScript
ECMA-262 Syntax Trees

Which of the 53 different nodes types are worth examining?

**Binding Pattern**
- ArrayPattern
- AssignmentPattern
- BindingPattern
- RestElement
- ObjectPattern

**Expression**
- ThisExpression
- Identifier
- Literal
- ArrayExpression
- SpreadElement
- ObjectExpression
- Property
- FunctionExpression
- ArrowFunctionExpression
- ClassExpression
- ClassBody
- MethodDefinition
- TaggedTemplateExpression
- TemplateElement
- TemplateLiteral
- MemberExpression
- Super
- Meta-Property
- NewExpression
- CallExpression
- UpdateExpression
- UnaryExpression
- BinaryExpression
- LogicalExpression
- ConditionalExpression
- YieldExpression
- AssignmentExpression
- SequenceExpression

**Statement**
- BlockStatement
- BreakStatement
- ContinueStatement
- DebuggerStatement
- DoWhileStatement
- EmptyStatement
- ExpressionStatement
- ForStatement
- ForInStatement
- ForOfStatement

**Imports**
- ImportDeclaration
- ImportSpecifier
- ImportDefaultSpecifier
- ImportNamespaceSpecifier
- ExportAllDeclaration
- ExportDefaultDeclaration
- ExportNamedDeclaration
Which nodes types are worth examining?

- We determined the **topmost node types in the patches** found by random search
- We determined the **frequency** with which node types appear in JavaScript programs
  - We have performed a study using \(~34,000\) JavaScript programs from the NPM repository
- We have **calculated a ratio** favoring high-frequency nodes that appear as topmost
  - Set a minimum threshold that limits which node types are examined by the local search
JavaScript AST

18 most worth node types for JavaScript source code size reduction

**Binding Pattern**
- ArrayPattern
- AssignmentPattern
- BindingPattern
- RestElement
- ObjectPattern

**Expression**
- ThisExpression
- Identifier
- Literal
- ArrayExpression
- SpreadElement
- ObjectExpression
- Property
- FunctionExpression
- ArrowFunctionExpression
- ClassExpression
- ClassBody
- MethodDefinition
- TaggedTemplateExpression
- TemplateElement
- TemplateLiteral
- MemberExpression

**Statement**
- BlockStatement
- BreakStatement
- ContinueStatement
- DebuggerStatement
- DoWhileStatement
- EmptyStatement
- ExpressionStatement
- ForStatement
- ForInStatement
- ForOfStatement

**FunctionDeclaration**
- IfStatement
- LabeledStatement
- ReturnStatement
- SwitchStatement
- SwitchCase
- ThrowStatement
- TryStatement
- CatchClause
- VariableDeclaration
- VariableDeclarator
- WhileStatement
- WithStatement

**Imports**
- ImportDeclaration
- ImportSpecifier
- ImportDefaultSpecifier
- ImportNamespaceSpecifier
- ExportAllDeclaration
- ExportDefaultDeclaration
- ExportNamedDeclaration
Which nodes types are worth examining?

- We examine all occurrences of each node type in a First-Ascent HC fashion
  - For small instances, we use all node types and reduce the search space to 89%
  - For larger ones, we discard `MemberExpression` and `Identifier` node types, reducing the space to 34%
  - This allows navigating the space several times in a reasonable time frame, even for large instances
Preliminary results: achieved reduction

<table>
<thead>
<tr>
<th>Program</th>
<th>RD</th>
<th>FAHC</th>
</tr>
</thead>
<tbody>
<tr>
<td>browserify</td>
<td>0.19</td>
<td>25.39</td>
</tr>
<tr>
<td>exactimer</td>
<td>2.06</td>
<td>26.76</td>
</tr>
<tr>
<td>jquery</td>
<td>0.19</td>
<td>79.89</td>
</tr>
<tr>
<td>lodash</td>
<td>0.33</td>
<td>6.23</td>
</tr>
<tr>
<td>minimist</td>
<td>0.14</td>
<td>2.68</td>
</tr>
<tr>
<td>plivo-node</td>
<td>0.58</td>
<td>33.24</td>
</tr>
<tr>
<td>pug</td>
<td>3.16</td>
<td>39.17</td>
</tr>
<tr>
<td>tleaf</td>
<td>3.81</td>
<td>67.07</td>
</tr>
<tr>
<td>underscore</td>
<td>0.30</td>
<td>10.10</td>
</tr>
<tr>
<td>uuid</td>
<td>1.05</td>
<td>23.60</td>
</tr>
<tr>
<td>xml2js</td>
<td>0.14</td>
<td>-2.78</td>
</tr>
</tbody>
</table>

- A huge difference from former results
- Some results are within an expected range
- Other results ... well, not so much!
- Some results are even curious ...

But they all pass all test cases! And we have at least 90% coverage!
Is this any different to dead code removal?

In some cases, not really.

A function from the d3-node library which is not exercised by test cases and was removed by the optimizer.
Is this any different to dead code removal?

But in other cases, yes it does.

Bitwise operation from the `uuid` library that had no effect on test cases, despite being covered by the test suite.
Can we help to improve tests or code review?

There seems to be an opportunity to co-evolve test cases and the code.

"Summertime testing" in the execute() library. All test cases use sorted data.
Can we help to improve tests or code review?

“A program does what the programmer commands, not necessarily what the programmer wants.”

By showing what it can destroy, optimization can help developers put their assumptions into solid test suites ... and close the gap.
What is next?

- We are compiling the numbers, strengthening the arguments, and hope to have a complete version of a paper with our results soon.
Thank you!