# Equivalence Testing for JavaScript Transformations

**Johannes Kloos**, Rupak Majumdar, Frank McCabe

October 23, 2015

# Reducing JavaScript file sizes

JavaScript programs often send a lot of unnecessary code.
We would like to eliminate "dead code" from pages, to send smaller JavaScript files.

# Reducing JavaScript file sizes

JavaScript programs often send a lot of unnecessary code.
We would like to eliminate "dead code" from pages, to send smaller JavaScript files.

The DOLOTO tool tries to do this.

# Reducing JavaScript file sizes

JavaScript programs often send a lot of unnecessary code.
We would like to eliminate "dead code" from pages, to send smaller JavaScript files.

The DOLOTO tool tries to do this.

Getting the transformation right turns out to be hard...

# How the transformation works

The transformation replaces functions that are not used immediately with “stub code” that loads the actual implementation from the network.

# How the transformation works

The transformation replaces functions that are not used immediately with "stub code" that loads the actual implementation from the network.

```
function func(arg) {
    ... /* lots and lots of code */
}
```

becomes (roughly)

```
function func_AZ123(arg) {
    /* get('x') loads body of x as string */
    func_AZ123 = eval(get('func'));
    return func_AZ123(arguments);
}
function func(arg) {
    return func_AZ123(arguments);
}
```

# What went wrong?

Excerpt of code from the site:

```
c = /xyz/.test(function() { xyz }) ?
    /\b_super\b/ : /.*/
```

Before running the transformation, this is equivalent to:

```
c = /\b_super\b/
```

But after running the transformation, we get:

```
c = /.*/
```

Note that `/xyz/.test(s)` implicitely transforms `s` to a string!

# JavaScript challenges

This work is based on a commercial implementation of the transformation.

It has to deal with:

- Correct variable scopes
- `toString` shenanigans
- Handling of anonymous and nested functions
- ...

# JavaScript challenges

This work is based on a commercial implementation of the transformation.

It has to deal with:

- Correct variable scopes
- `toString` shenanigans
- Handling of anonymous and nested functions
- ...

Are we doing it right?

# JavaScript challenges

This work is based on a commercial implementation of the transformation.

It has to deal with:

- Correct variable scopes
- `toString` shenanigans
- Handling of anonymous and nested functions
- ...

Are we doing it right?
Have we even covered everything?

# Verifying the transformation – the problem

We want to make sure that the transformation changes pages in such a way that the change is invisible to the user.
Questions:

- What does “invisible to the user” actually mean?
- How do we verify that?

# Verifying the transformation – the problem

We want to make sure that the transformation changes pages in such a way that the change is invisible to the user.
Questions:

- What does "invisible to the user" actually mean?
  More on this later!
- How do we verify that?

# Verifying the transformation – the problem

We want to make sure that the transformation changes pages in such a way that the change is invisible to the user.

Questions:

- What does "invisible to the user" actually mean?
  More on this later!
- How do we verify that?
  One possible approach:
  "Just run the interpreter test suite" – we tried that.
  Not sufficient, doesn't catch the `toString` problem

# Another approach: Formal verification

Idea: Use formal, mathematical proof that the transformation is always correct, like in CompCert.

# Another approach: Formal verification

Idea: Use formal, mathematical proof that the transformation is always correct, like in CompCert.

JavaScript semantics make this extremely hard; consider `eval`, `Function.toString`, the `Function` constructor, ....

# Another approach: Formal verification

Idea: Use formal, mathematical proof that the transformation is always correct, like in CompCert.

JavaScript semantics make this extremely hard; consider `eval`, `Function.toString`, the `Function` constructor, . . . .

# Another idea: CSmith

The question that CSmith attempts to answer: How well do exisiting C compilers follow the language specification?
Idea:

1. Generate many small test programs (with only clearly defined behavior)

# Another idea: CSmith

The question that CSmith attempts to answer: How well do exisiting C compilers follow the language specification?
Idea:

1. Generate many small test programs (with only clearly defined behavior)
2. Compile each program with multiple compilers.

# Another idea: CSmith

The question that CSmith attempts to answer: How well do exisiting C compilers follow the language specification?
Idea:

1. Generate many small test programs (with only clearly defined behavior)
2. Compile each program with multiple compilers.
3. Run all programs. For each program declare the most common output as “correct”; all other versions are considered mis-compiled.

# Another idea: CSmith

The question that CSmith attempts to answer: How well do exisiting C compilers follow the language specification?
Idea:

1. Generate many small test programs (with only clearly defined behavior)
2. Compile each program with multiple compilers.
3. Run all programs. For each program declare the most common output as "correct"; all other versions are considered mis-compiled.

# Another idea: CSmith

The question that CSmith attempts to answer: How well do exisiting C compilers follow the language specification?
Idea:

1. Generate many small test programs (with only clearly defined behavior)
2. Compile each program with multiple compilers.
3. Run all programs. For each program declare the most common output as "correct"; all other versions are considered mis-compiled.

Results: Many bona-fide bugs were found in all tested C compilers; false positives not a problem.

# Another idea: CSmith

The question that CSmith attempts to answer: How well do exisiting C compilers follow the language specification?
Idea:

1. Generate many small test programs (with only clearly defined behavior)
2. Compile each program with multiple compilers.
3. Run all programs. For each program declare the most common output as "correct"; all other versions are considered mis-compiled.

Results: Many bona-fide bugs were found in all tested C compilers; false positives not a problem.
Question: Can we apply a similar testing approach?

# Our chosen approach: Model-based testing

Based on CSmith.

# Our chosen approach: Model-based testing

Based on CSmith.
Adapted approach:

1. Generate random JavaScript programs ("unmodified programs")

# Our chosen approach: Model-based testing

Based on CSmith.
Adapted approach:

1. Generate random JavaScript programs ("unmodified programs")
   So far, we use existing test suites. Some pitfalls mentioned later.

# Our chosen approach: Model-based testing

Based on CSmith.
Adapted approach:

1. Generate random JavaScript programs ("unmodified programs")
   So far, we use existing test suites. Some pitfalls mentioned later.
2. Run unmodified programs through the transformation
   This gives "transformed programs".

# Our chosen approach: Model-based testing

Based on CSmith.
Adapted approach:

1. Generate random JavaScript programs ("unmodified programs")
   So far, we use existing test suites. Some pitfalls mentioned later.
2. Run unmodified programs through the transformation
   This gives "transformed programs".
3. Compare results of running unmodifed and transformed programs.
   The rest of the talk focuses on this!

# Outline

- What are valid test cases?
- Overview of the result comparison
- Experimental results

# Outline

- What are valid test cases?
- Overview of the result comparison
- Experimental results

# Not every JavaScript program is a test case!

```
// Array containing expected property names of the
// window object, i.e. the names described in the
// ECMAScript standard and by the DOM.
var exp = new Array (...);
// The names that are actually contained.
var real = Object.getOwnPropertyNames(window);
// Names that were not expected
var extra = exp.filter(function (name) {
  return Array.prototype.indexOf(real, name) == -1;
});
console.log("Letter-of-the-law compliant: " +
   (extra.length = 0));
```

# Not every JavaScript program is a test case!

```
// Array containing expected property names of the
// window object, i.e. the names described in the
// ECMAScript standard and by the DOM.
var exp = new Array (...);
// The names that are actually contained.
var real = Object.getOwnPropertyNames(window);
// Names that were not expected
var extra = exp.filter(function (name) {
  return Array.prototype.indexOf(real, name) == -1;
});
console.log("Letter-of-the-law compliant: " +
    (extra.length = 0));
```

This code would break when transformed!

# Not every JavaScript program is a test case!

```
// Array containing expected property names of the
// window object, i.e. the names described in the
// ECMAScript standard and by the DOM.
var exp = new Array (...);
// The names that are actually contained.
var real = Object.getOwnPropertyNames(window);
// Names that were not expected
var extra = exp.filter(function (name) {
  return Array.prototype.indexOf(real, name) == -1;
});
console.log("Letter-of-the-law compliant: " +
   (extra.length = 0));
```

This code would break when transformed!

Reason: This code performs very heavy introspection.

# Introspection in JavaScript

Introspection is widespread:

- Feature checks:

```
if (Array.prototype.filter == null) { /* apply
    polyfill */ }
```

# Introspection in JavaScript

Introspection is widespread:

- Feature checks:

```
if (Array.prototype.filter == null) { /* apply
    polyfill */ }
```

- Object-oriented hacks using `Function.toString`:

```
if (/super/.test(f.toString)) { /* provide
   super argument */ }
```

# Introspection in JavaScript

Introspection is widespread:

- Feature checks:

```
if (Array.prototype.filter == null) { /* apply
    polyfill */ }
```

- Object-oriented hacks using `Function.toString`:

```
if (/super/.test(f.toString)) { /* provide
    super argument */ }
```

- `for..in`
- `Object.getOwnProperty` and the like

# The open world assumption

In a web application, many scripts and libraries must co-exist. This is an open world: Things can be added to the environment without our knowledge.

# The open world assumption

In a web application, many scripts and libraries must co-exist. This is an open world: Things can be added to the environment without our knowledge.

Therefore, a script must not make assumptions about parts of the environment that it does not control.

# The open world assumption

In a web application, many scripts and libraries must co-exist. This is an open world: Things can be added to the environment without our knowledge.

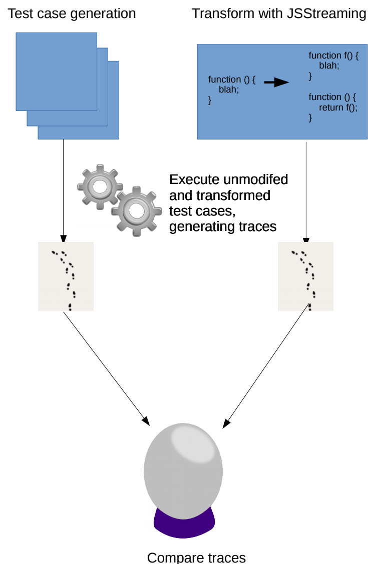
Therefore, a script must not make assumptions about parts of the environment that it does not control.

In particular, only tame introspection: Only apply introspection to objects it controls (except some feature checks).

# Outline

- What are valid test cases?
- Overview of the result comparison
- Experimental resuls

# The process in a picture



# What is a trace?

A trace is a sequence of events that describe the execution of a program.
Some of the events we consider:

| | |
|---|---|
| Read(*ref*, *val*) | Read value *val* from *ref* |
| Write(*ref*, *val*) | Write value *val* to *ref* |
| FunPre(*f*, *this*, *args*) | Call function *f* on *this* with arguments *args* |

# What is a trace?

A trace is a sequence of events that describe the execution of a program.
Some of the events we consider:

| | |
|---|---|
| Read(*ref*, *val*) | Read value *val* from *ref* |
| Write(*ref*, *val*) | Write value *val* to *ref* |
| FunPre(*f*, *this*, *args*) | Call function *f* on *this* with arguments *args* |

We also track variable declarations, exception handling, expression evaluation, . . . .

# An example of a trace

```
console.log("x")
```

gives

Read(console, obj:42)

Read(obj:42.log, func:23)

Literal("x")

FunPre(func:23, obj:42, obj:105) // obj:105 looks like `["x"]`

...

# A simple equivalence criterion

Declare some events as externally visible:

- Calls to external functions.
  A function is external if it is implemented using native code.
- Writes to global variables and objects.
  A variable/object is global if it is not declared/created inside the script.

Then two traces can be defined as equivalent when the subsequences consisting of global events are the same.

# An observation on trace structures

| Original trace | Transformed trace |
|---|---|
| $e_1$ | $e_1$ |
| | $\mathrm{init}_1$ |
| | $\mathrm{init}_2$ |
| $e_2$ | $e_2$ |
| | $\mathrm{init}_3$ |
| $e_3$ | $e_3$ |
| call | call |
| | enter |
| | $\mathrm{wrap}_1$ |
| | call |
| enter | enter |
| $e_4$ | $e_4$ |
| $e_5$ | $e_5$ |
| exit | exit |
| | return |
| | exit |
| return | return |
| $e_6$ | $e_6$ |

# Our final equivalence criterion

Basically, two traces $A$ and $B$ are equivalent (for the transformation) if $A$ is a sub-trace of $B$ "in a nice way".

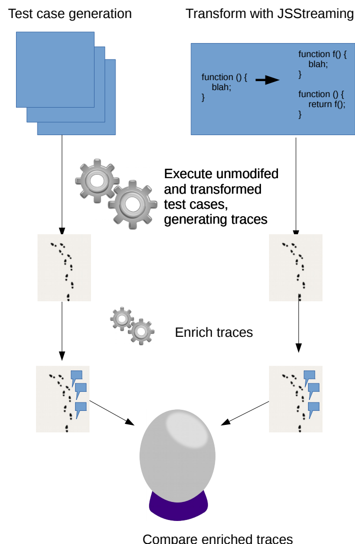
More formally: Two traces $A$ and $B$ are equivalent (for the transformation) if:

- $B$ can be divided into subsequences consisting of paired events, initialization events and wrapper events.
- The events in $A$ correspond to the subsequence of paired events in $B$[1].
- Wrapper and initialization events allow only events that are not externally visible.
- Restrictions on how to split trace $B$: See paper.

[1] Up to renaming of memory cells and the like

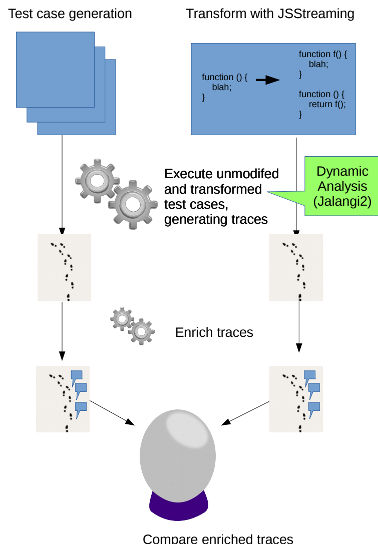
# Sketching the main steps

The analysis consists of three main steps:



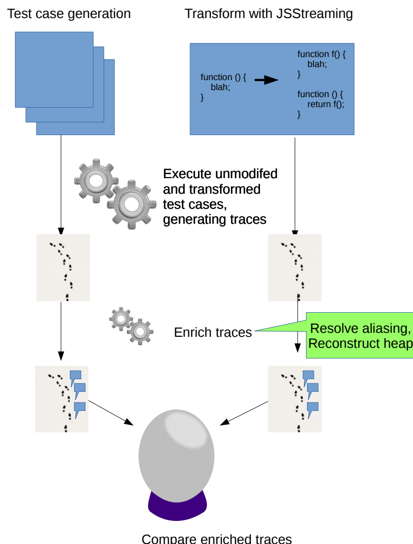
# Sketching the main steps

The analysis consists of three main steps:



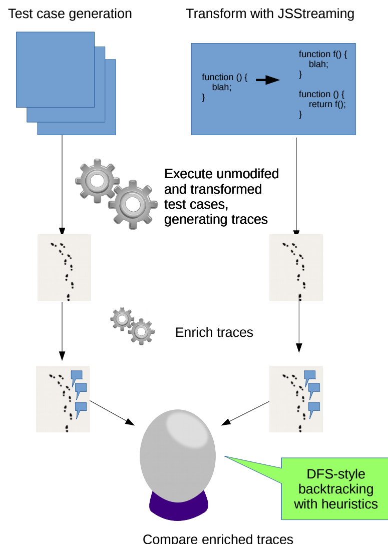
# Sketching the main steps

The analysis consists of three main steps:



# Sketching the main steps

The analysis consists of three main steps:

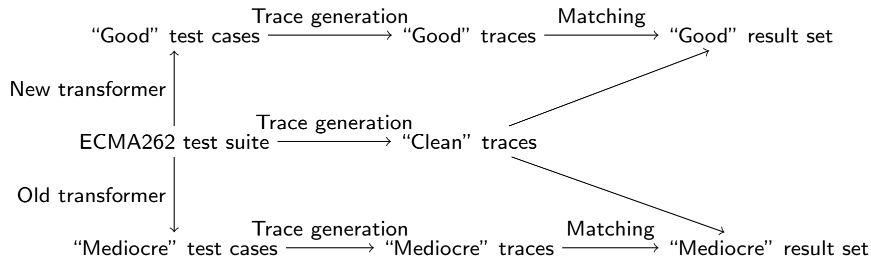


# Outline

- What are valid test cases?
- Overview of the result comparison
- Experimental results

# Trying it out

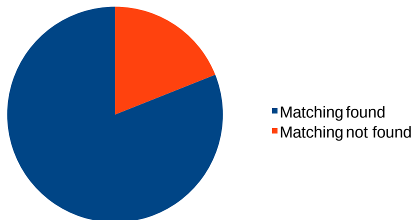
First round of experimental results: Run on test cases from the ECMA262 test suite.




Two questions:

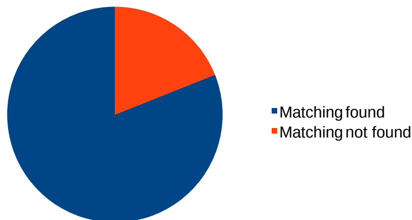
1. How stable is testing? How many false positives/false negatives do we get?
2. How well does error detection work?

# Stability



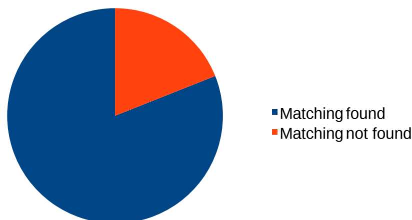

In total: 4972 test cases; 3883 found matches.

# Stability




In total: 4972 test cases; 3883 found matches.
Observations: No false positives found (and a sample of traces looks fine), but the rate of potential false negatives is still pretty high.

# Stability

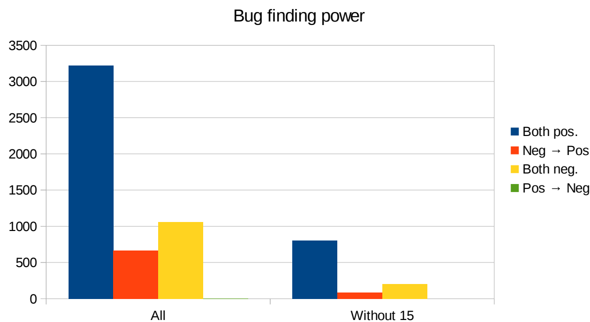



In total: 4972 test cases; 3883 found matches.

Observations: No false positives found (and a sample of traces looks fine), but the rate of potential false negatives is still pretty high.

Further analysis: centered around handling of higher-order functions ("chapter 15") and introspection/reflection.

# Error detection




The additional non-matches (red bars) point to actual errors!

# Summary

We are working on verifying a JavaScript source-to-source transformation through systematic testing.

The test approach is different from traditional testing: We decouple the decision whether a test case has passed from the test input.

This approach seems to be working reasonably well for for our transformation; can it be applied for other transformations/applications?

# How we do trace collection

Jalangi2: Dynamic analysis framework for JavaScript. It Works by instrumenting JavaScript code.

We simply dump all events. Some special handling to give objects a unique identifier.

Outputs a JSON file containing:

- The actual trace.
- The initial state of each object.
- The body of all functions, as string (if internal) or with unique ID (if external).
- Information about global objects.

# What a refined trace file contains

Refined trace files are very similar to traces from trace collection.
Main differences:

- The trace is modified to resolve all aliasing in reads and writes.
- Variable shadowing is eliminated using a SSA-like transform.
- Information is introduced to provide a snapshot of the heap after each step.

# An example of an obstacle

```
function f(x,y) {
  x=42;
  y=42;
  console.log(arguments[0] + "," + arguments[1]);
}
f(1,2);
f(1);
```

Output:

# An example of an obstacle

```
function f(x,y) {
  x=42;
  y=42;
  console.log(arguments[0] + "," + arguments[1]);
}
f(1,2);
f(1);
```

Output:

```
42, 42
42, undefined
```

# How we match traces

Basically, split the transformed trace using a backtracking algorithm.

On the trace level, various heuristics for fast search.

The algorithm has to match objects; this uses the heap information from trace enrichment.