Static Analysis for JavaScript-Style Eval

Dr Martin Lester

University of Oxford

2016–08–08

Motivation

Web applications written in the dynamically typed language JavaScript regularly handle sensitive data.

- Reasoning about their behaviour is an important security problem.
- JavaScript is a difficult language to reason about.

Why is JavaScript difficult?

- Many features, including **eval**, which is widely used.
- **eval** takes a string, interprets it as a piece of code, then executes that code.
- We can think of **eval** as a form of metaprogramming.

What can we do about **eval**?

Analysing Eval

Analysing **eval** is hard!

- The **eval** construct takes a string and executes it as code.
- Its behaviour can be so variable that static analysis seems utterly hopeless.
- Metaprogramming is in general poorly understood.

Naive approach:

- Use a string analysis to work find out all the string values that might be **eval**ed, then analyse the code strings.
- Fine if you only have finitely many code strings.
- Otherwise doomed.

```
$x = "2"$
while $(f())$
    $x = "2 *" + x$
eval $x$
```

Staged Metaprogramming

How can we make **eval** easier to analyse statically?

- A code string is difficult to analyse because it has no structure.
- In practice, programs construct **eval**ed code mainly by splicing together code templates.

The staged metaprogramming formalism introduces three primitives to capture this:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

The Boxing Algorithm

The Boxing Algorithm provides an automated transformation from **eval** to staged metaprogramming.

The basic idea is that we transform:

- code constants into **box** expressions;
- concatenation of code strings into splicing using **unbox**;
- **eval** into **run**.

For example:	becomes:
$\mathbf{let}\ x = \texttt{"y"}\ \mathbf{in}$	$\mathbf{let}\ x = \mathbf{box}\ y\ \mathbf{in}$
$\mathbf{eval}\ x$	$\mathbf{run}\ x$

while:	becomes:
$\mathbf{let}\ f = \mathbf{fun}(z)\{3 * z\}\ \mathbf{in}$	$\mathbf{let}\ f = \mathbf{fun}(z)\{3 * z\}\ \mathbf{in}$
$\mathbf{let}\ y = \texttt{"2"}\ \mathbf{in}$	$\mathbf{let}\ y = \mathbf{box}\ 2\ \mathbf{in}$
$\mathbf{let}\ x = \texttt{"f("} + y + \texttt{")"}\ \mathbf{in}$	$\mathbf{let}\ x = \mathbf{box}(f(\mathbf{unbox}\ y))\ \mathbf{in}$
$\mathbf{eval}\ x$	$\mathbf{run}\ x$

Outline

The basic idea is simple enough:

- Assume **eval** does nothing.
- Use a string analysis to work out which strings get **eval**ed or concatenated to form new code values.
- Parse the strings and replace with **box** expressions.
- Replace concatenation with use of **unbox**.

However, there are many complications:

- Concatenation can change how a string is lexed/parsed.
- We need to be able to parse "incomplete" expressions containing holes.
- The **eval**ed code could introduce new uses of **eval**, or new code strings . . .
- . . . so we need to repeat the analysis with the transformed program
- . . . (until we eventually reach a fixed point or get stuck)
- . . . which means the string analysis needs to work with staged metaprogramming.

Outline

The basic idea is simple enough:

- Assume **eval** does nothing.
- Use a string analysis to work out which strings get **eval**ed or concatenated to form new code values.
- Parse the strings and replace with **box** expressions.
- Replace concatenation with use of **unbox**.

However, there are many complications:

- Concatenation can change how a string is lexed/parsed.
- We need to be able to parse "incomplete" expressions containing holes.
- The **eval**ed code could introduce new uses of **eval**, or new code strings . . .
- . . . so we need to repeat the analysis with the transformed program
- . . . (until we eventually reach a fixed point or get stuck)
- . . . which means the string analysis needs to work with staged metaprogramming.

Thanks for listening. Any questions?