Detecting Redundant CSS Rules in HTML5 Applications: A Tree Rewriting Approach

Anthony W. Lin (with Matthew Hague and Luke Ong)

Yale-NUS College, Singapore
Royal Holloway, University of London, UK
Oxford University, UK
A webpage has three main components...
Introduction to Webpages

Super Heroes

The Document Object Model (DOM) tree.
A dynamic / scripting component.
A dynamic / scripting component.
Super Heroes

The CSS component.
The CSS Redundancy Problem

CSS Stylesheets can become very large.

- Usual development bloat (e.g. plugins with generic stylesheets).
  - E.g. Nivo-Slider has 172 selectors, 131 redundant in demo above.
The CSS Redundancy Problem

CSS Stylesheets can become very large.

- Usual development bloat (e.g. plugins with generic stylesheets).
  - E.g. Nivo-Slider has 172 selectors, 131 redundant in demo above. 60% of them are redundant... [Mesbah and Mirshokraie]
- 30% of rendering time is spent on selectors [Meyerovich and Bodik].
Existing Solutions

Existing tools for cleaning CSS are quite limited:

- Cilla [Mesbah and Mirshokraie] and UnCSS [Martino]
  - Explores as much of a page as it can.
  - Reports which selectors were not used.
  - Unsound.
Existing solutions for cleaning CSS are quite limited:

- Cilla [Mesbah and Mirshokraie] and UnCSS [Martino]
  - Explores as much of a page as it can.
  - Reports which selectors were not used.
  - Unsound.
- Both of these approaches can remove useful CSS rules.
  - This can ruin the look of your website.
Existing Solutions

Existing tools for cleaning CSS are quite limited: Cilla [Mesbah and Mirshokraie] and UnCSS [Martino] explore as much of a page as they can. They report which selectors were not used. Unsound. Both of these approaches can remove useful CSS rules. This can ruin the look of your website.
Existing Solutions

Existing tools for cleaning CSS are quite limited: Cilla [Mesbah and Mirshokraie] and UnCSS [Martino].

- Explores as much of a page as it can.
- Reports which selectors were not used.
- Unsound.

Both of these approaches can remove useful CSS rules. This can ruin the look of your website.
Existing Solutions

Existing tools for cleaning CSS are quite limited:

- Cilla [Mesbah and Mirshokraie] and UnCSS [Martino]
  - Explores as much of a page as it can.
  - Reports which selectors were not used.
  - Unsound.
  - Both of these approaches can remove useful CSS rules.
  - This can ruin the look of your website.
Existing tools for cleaning CSS are quite limited:

- Cilla [Mesbah and Mirshokraie] and UnCSS [Martino]
  - Explores as much of a page as it can.
  - Reports which selectors were not used.
  - Unsound.

Both of these approaches can remove useful CSS rules.

- This can ruin the look of your website.
- And break functionality (UnCSS breaks Nivo-Slider).
Static Analysis?
Static Analysis?

- Leading tools: WALA and TAJS
Static Analysis?

- Leading tools: WALA and TAJS
- Using them to identify redundant CSS is challenging.
Static Analysis?

- Leading tools: WALA and TAJS
- Using them to identify redundant CSS is challenging.
- **Problem 1**: Call graph of JS+jQuery is hard to construct
Static Analysis?

- Leading tools: WALA and TAJS
- Using them to identify redundant CSS is challenging.
  - Problem 1: Call graph of JS+jQuery is hard to construct
  - Problem 2: DOM tree is not precisely tracked
Static Analysis?

- Leading tools: WALA and TAJS
- Using them to identify redundant CSS is challenging.
- **Problem 1**: Call graph of JS+jQuery is hard to construct
- **Problem 2**: DOM tree is not precisely tracked

See *Andreasen and Moller’14* for an up-to-date survey on static analysis of JavaScript+jQuery.
Our Contributions

Clean tree-rewriting model of DOM dynamics due to JS.

- Precise
- Automatic analysis (via reduction to symbolic pushdown systems)
Our Contributions

Clean tree-rewriting model of DOM dynamics due to JS.
- Precise
- Automatic analysis (via reduction to symbolic pushdown systems)

Proof-of-concept translation from HTML5 to our model.
- Models jQuery functions in how they modify the DOM.
- Does not yet support many features of JavaScript.
Our Contributions

Clean tree-rewriting model of DOM dynamics due to JS.
- Precise
- Automatic analysis (via reduction to symbolic pushdown systems)

Proof-of-concept translation from HTML5 to our model.
- Models jQuery functions in how they modify the DOM.
- Does not yet support many features of JavaScript.

Promising experimental results.
Our approach (in a nutshell)

- Aim to **overapproximate** DOM dynamics with tree-rewriting model $\mathcal{R}$
Our approach (in a nutshell)

- Aim to overapproximate DOM dynamics with tree-rewriting model $\mathcal{R}$
- Redundant node selectors in $\mathcal{R} \implies$ redundant in HTML5
Our Tree-Rewriting Model: Domain

DOM is abstracted as an (unordered) unranked tree with class labels.
DOM is abstracted as an (unordered) unranked tree with class labels.

HTML elements and node IDs are treated as constant classes
A rewrite rule is a pair \((g, \chi)\), where:

- \(g\) is a “guard” (a.k.a. “node selector”): modal logic formula with modalities \(\langle \uparrow \rangle\), \(\langle \uparrow^+ \rangle\), \(\langle \downarrow \rangle\), \(\langle \downarrow^+ \rangle\),

- \(\chi\) is a rewrite operation: AddClass, AddChild, RemoveClass, RemoveNode.
A rewrite rule is a pair \((g, \chi)\), where:

- \(g\) is a “guard” (a.k.a. “node selector”): modal logic formula with modalities \(⟨↑⟩, ⟨↑^+⟩, ⟨↓⟩, ⟨↓^+⟩\),
- \(\chi\) is a rewrite operation: AddClass, AddChild, RemoveClass, RemoveNode.

A tree-rewrite system is a finite set of rewrite rules.
A jQuery line

\[
\$(\".info\\).appendChild(\"<div class='comic'>DC Comics</div>\")
\]

is represented by

\[
(info, AddChild(div comic))
\]
A jQuery line

```javascript
$(".info").appendChild("<div class='comic'>DC Comics</div>"
```

is represented by

```javascript
(info, AddChild(div comic))
```
Examples

A CSS Selector

\[ \text{.info} > .date \{ \text{font-style: italic; } \} \]

can be represented by

\[ (\text{date} \land \langle \uparrow \rangle \text{info}, \text{AddClass(cssrule1)}) \]
Operational Semantics of TRS

\[ T_1 \rightarrow_\mathcal{R} T_2 \text{ if } T_1 \text{ can be rewritten into } T_2 \]

i.e. \( \exists \) a node \( v \) in \( T_1 \) where some rule \((g, \chi)\) in \( \mathcal{R} \) can be “fired”.
Redundancy Problem

**INPUT**: a TRS $\mathcal{R}$, an initial DOM tree $T$, and a $S$ set of node selectors

**QUESTION**: Identify selectors in $S$ that cannot be matched (in all reachable trees)
Solving the Redundancy Problem

Q: given a tree, rules and classes, which classes are redundant?
Q: given a tree, rules and classes, which classes are redundant?

CSS redundancy is **undecidable** in general.
Solving the Redundancy Problem

Q: given a tree, rules and classes, which classes are redundant?

CSS redundancy is **undecidable** in general

In practice, suffices to restrict to positive guards
Solving the Redundancy Problem

Q: given a tree, rules and classes, which classes are redundant?

CSS redundancy is undecidable in general.

In practice, suffices to restrict to positive guards.

e.g. date \land \langle \uparrow \rangle info
Q: given a tree, rules and classes, which classes are redundant?

CSS redundancy is **undecidable** in general

In practice, suffices to restrict to positive guards

e.g. \texttt{date} \land \langle \uparrow \rangle \texttt{info}

**Theorem**: The redundancy problem for positive TRS is efficiently reducible to analysis of symbolic PDS, which is EXP-complete, but for which fast solvers exist.
The Reduction

**Theorem**: The redundancy problem for positive TRS is efficiently reducible to analysis of symbolic PDS, which is EXP-complete, but for which fast solvers exist.

- One boolean variable for each class
The Reduction

**Theorem**: The redundancy problem for positive TRS is efficiently reducible to analysis of symbolic PDS, which is EXP-complete, but for which fast solvers exist.

- One boolean variable for each class
- Stack used in a “depth-first search” of the tree.
The Reduction

**Theorem:** The redundancy problem for positive TRS is efficiently reducible to analysis of symbolic PDS, which is EXP-complete, but for which fast solvers exist.

- One boolean variable for each class
- Stack used in a “depth-first search” of the tree.
- New nodes can be created by pushing onto the stack.
The Reduction

**Theorem:** The redundancy problem for positive TRS is efficiently reducible to analysis of symbolic PDS, which is EXP-complete, but for which fast solvers exist.

- One boolean variable for each class
- Stack used in a “depth-first search” of the tree.
- New nodes can be created by pushing onto the stack.
- Backtrack by popping.
The Reduction

**Theorem:** The redundancy problem for positive TRS is efficiently reducible to analysis of symbolic PDS, which is EXP-complete, but for which fast solvers exist.

- One boolean variable for each class
- Stack used in a “depth-first search” of the tree.
- New nodes can be created by pushing onto the stack.
- Backtrack by popping.
- (Since guards are positive, we can always recreate nodes.)
Implementation

We implemented a tool TreePed to test the approach.

- A rough tool for extracting rules from jQuery script.
- jMoped used as a pushdown backend.

Tested on a number of examples (next slide).

- Reasonable run times.
- Identified all and only redundant rules.
- Two real-world examples, five made up examples.
## Results Table

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Ns</th>
<th>Ss</th>
<th>Ls</th>
<th>Rs</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bikes.html</td>
<td>22</td>
<td>18 (0)</td>
<td>97</td>
<td>37</td>
<td>3.6s</td>
</tr>
<tr>
<td>comments.html</td>
<td>5</td>
<td>13 (1)</td>
<td>43</td>
<td>26</td>
<td>2.9s</td>
</tr>
<tr>
<td>example.html</td>
<td>11</td>
<td>1 (0)</td>
<td>28</td>
<td>4</td>
<td>.6s</td>
</tr>
<tr>
<td>example-up.html</td>
<td>8</td>
<td>1 (1)</td>
<td>15</td>
<td>3</td>
<td>.6s</td>
</tr>
<tr>
<td>igloo/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>index.html</td>
<td>145</td>
<td></td>
<td>24</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>engineering.html</td>
<td>236</td>
<td></td>
<td>24</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Nivo-Slider/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>demo.html</td>
<td>15</td>
<td>172 (131)</td>
<td>501</td>
<td>21</td>
<td>6.3s</td>
</tr>
<tr>
<td>transactions.html</td>
<td>19</td>
<td>9 (0)</td>
<td>37</td>
<td>6</td>
<td>1.6s</td>
</tr>
</tbody>
</table>

Ns — # of HTML elements in the initial tree  
Ss — # of CSS rules (redundant CSS rules)  
Ls — # of lines of JavaScript (cloc)  
Rs — # of rules extracted from JS
Summary and Future Work

Web pages are dynamic programs:

- Manipulate a tree data structure (DOM).
- Can be modelled by tree rewrite systems.
- Model-checking can be used for optimising CSS rules.

Future work:

- Systematically extracting rewrite rules from JavaScript.
  - JavaScript analysis is hard! (c.f. Moller et al)