Automated Refactoring for Size Reduction of CSS Style Sheets

Martí Bosch, Pierre Genevès, Nabil Layaïda

Automated Refactoring for Size Reduction of CSS Style Sheets

Martí Bosch
UPC & Inria
marti.bosch-padros@inria.fr

Pierre Genevès
CNRS
pierre.geneves@cnrs.fr

Nabil Layaïda*
Inria
nabil.layaida@inria.fr

ABSTRACT

Cascading Style Sheets (CSS) is a standard language for stylizing and formatting web documents. Its role in web user experience becomes increasingly important. However, CSS files tend to be designed from a result-driven point of view, without much attention devoted to the CSS file structure as long as it produces the desired results. Furthermore, the rendering intended in the browser is often checked and debugged with a document instance. Style sheets normally apply to a set of documents, therefore modifications added while focusing on a particular instance might affect other documents of the set.

We present a first prototype of static CSS semantical analyzer and optimizer that is capable of automatically detecting and removing redundant property declarations and rules. We build on earlier work on tree logics to locate redundancies due to the semantics of selectors and properties. Existing purely syntactic CSS optimizers might be used in conjunction with our tool, for performing complementary (and orthogonal) size reduction, toward the common goal of providing smaller and cleaner CSS files.

Categories and Subject Descriptors

1.7 [**Document and Text Processing**]: Document preparation; D.2 [**Software Engineering**]: Testing and Debugging

Keywords

Web development; Style sheets; CSS; Debugging

1. INTRODUCTION

Cascading style sheets (CSS) is one of the main components of web development, used to describe the aspect and format of a markup document, most of the time an HTML web page. The simplicity of its syntax makes it attractive for designers and amateurs as it only requires assigning values to properties under certain elements of the document.

However CSS language presents a set of combinatorial features that empower its possibilities, while inevitably making it more complex. A single document might use several style sheets, include embedded CSS under the `style` HTML element, or even use inline styles set directly as attributes of the element concerned. Furthermore, the language used to write selectors is very expressive. The combination of these features might lead to a series of semantical errors and redundancies that make it difficult to spot the origin of problems when one does not get the desired output in the browser. The tool we propose is aimed to help web developers in that matter, as well as reducing the style sheet size.

2. RELATED WORKS

In sharp contrast with existing debugging tools [5, 6, 7, 4], we propose a method for automatically analysing and refactoring a CSS file, with the guarantee that the rendering in the browser will not be affected, *for any possible document* that might use the CSS. A highly novel aspect of our tool is that it is capable of performing CSS refactoring by a semantical analysis of a given CSS file, in the absence of any other information (such as a particular document instance). For this purpose, we build on our previous logical modeling of CSS selectors introduced in [2]. The main difference between the present work and [2] is that [2] focuses on the detection of rendering bugs, whereas the present work seeks to perform automatic CSS size reduction while preserving the rendering semantics.

3. CSS ANALYSIS AND REFACTORING

In CSS, rules are encapsulated by a selector, that points to the set of elements that will be affected by the rule's declarations. We borrow ideas from the fields of set theory and tree logics to analyze the sets of elements pointed by the different selectors present in a CSS file. Our tool is concerned with the detection of semantical relations between CSS selectors. When some of these relations are detected, our tool might determine that a property declaration is unnecessary and it will thus be deleted, based on the *specificity* of selectors. In CSS, a selector's *specificity* is a vector of four integers (a, b, c, d), where $a = 1$ if the property is declared in a `style` attribute ($a = 0$ otherwise), b is the number of `id` attributes (of the form "#␣") in the selector, c is the number of other

* Detailed affiliation of authors: M. Bosch[1234], P. Genevès[213], N. Layaïda[123] with: [1]Inria; [2]CNRS, LIG; [3]Univ. Grenoble Alpes; [4]Universitat Politècnica de Catalunya. Work partly supported by ANR project Typex (ANR-11-BS02-007).

attributes and pseudo-classes in the selector, and d is the number of element names and pseudo-elements in the selector. Our tool exploits the facts that: (1) if selectors of two different rules have the same specificity, then the last rule in the style sheet gains precedence; (2) when several selectors point to the same set of elements, then the declarations under the one with higher *specificity* gain precedence.

3.1 Containment of selectors

One fundamental relation between two CSS selectors is *containment*. For example we say that "`ul > li`" is *contained* into "`li`" since any "`li`" element with an "`ul`" parent is indeed a "`li`" element. The existence of containment relations is determined by the analysis of the nested structures of elements and the sets of attributes carried by elements. A selector such as "`p.someclass`" is contained into "`p`", since any "`p`" element with "`class`" attribute "`someclass`" is indeed a "`p`" element. These two kinds of containment can occur simultaneously as it is the case with "`table td#someid`" $\subset$ "`td`". More generally, containment relates sets of pointed elements that are associated with selectors.

Given two selectors S_b and S_p, S_b is contained in S_p iff any element pointed by S_b is also pointed by S_p. In this section we treat only *proper containment*, which means $S_b \subset S_p$ and not $S_b \subseteq S_p$. Under these circumstances, for each property declared under both selectors, there are two different procedures according to the selector's specificity:

Refactoring 1. ***Subset more specific****: delete the property declaration from S_b only if it has the same value set under both S_b and S_p.*

Refactoring 2. ***Subset less specific****: delete the property declaration from S_b, since the value set under S_p will always override the one under S_b.*

For example, consider the following code snippet:

Listing 1: containment_input.css

```
1  table.foo { color: #333;
2              font-size: 12px;
3              font-weight: bold }
4
5  table { color: #666;
6          font-size: 12px }
```

Note that "`table.foo`" $\subset$ "`table`" and the subset has a higher specificity $(0, 0, 1, 1)$ against $(0, 0, 0, 1)$ from the superset, so we are in the case of *refactoring 1*. Consequently, we have to preserve the "`color: #333`" declaration as it will override the one from "`table`" when both rules apply. On the other hand, the "`font-size`" property statement can be removed from the subset as the same value is already pulled from the superset "`table`". The following code corresponds to the output of the analysis:

Listing 2: containment_output.css

```
1  table.foo { color: #333;
2              font-weight: bold }
3
4  table { color: #666;
5          font-size: 12px }
```

If "`table`" was more specific than "`table.foo`", *refactoring 2* would apply and the "`color`" property declaration could be erased from the subset too as its value would always be overridden by the dominant "`#666`" set in the superset.

3.2 Equivalence of selectors

Another relation between CSS selectors that can lead to some refactoring is *equivalence*. Two or more CSS selectors can be equivalent in several ways. Is not uncommon to find a rule R_i with a selector such as "`body`", and later in the same file another rule R_j with the same selector "`body`". Another case of equivalence could be a class selector "`.classname`" and the attribute selector "`[class='classname']`", or any similar case with the dual syntax for `id` attributes. These examples could be studied with basic string processing, but the logical and semantical analysis of selectors allows us to detect more complex equivalences too such as the one between selectors "`p:nth-child(odd), p:nth-child(even)`" and "`p`", as every paragraph is either odd or even.

Given two selectors S_i and S_j, they are *equivalent* iff any element pointed by S_i is also pointed by S_j and vice-versa. In this context there is only one procedure over the bodies:

Refactoring 3. *For each property declared under both selectors, delete the statements under the less specific selector.*

To illustrate the analysis, let's look at the next listing:

Listing 3: equivalence_input.css

```
1  div#bar { font-style: italic;
2            border: none }
3
4  div[id='bar'] { color: #666;
5                  font-style: normal;
6                  border: none }
```

In this case we have two equivalent selectors, "`div#bar`" and "`div[id='bar']`" whose specificities are $(0, 1, 0, 1)$ and $(0, 0, 1, 1)$ respectively. For the properties declared under both rules, the values from "`div#bar`" will dominate, so the ones under "`div[id='bar']`" will never apply, as selectors point to the same set of elements. This means that for "`font-style`" and "`border`", the declarations can be safely erased from "`div[id='bar']`", resulting in the following output:

Listing 4: equivalence_output.css

```
1  div#bar { font-style: italic;
2            border: none }
3
4  div[id='bar'] { color: #666 }
```

3.3 Inheritance of properties

Whenever containment or equivalence relations are detected between selectors, several selectors point to the same elements, and specificity decides which declarations get precedence. In the context of inheritance, an element might be pointed by only one selector and yet be affected by declarations outside the concerning rule. This is because the declaration has been propagated from some ancestor through the inheritance mechanism. Consider a selector using a descendant combinator "$E_{anc} > E_{desc}$" with a inheritable property

declaration $P_a : V_a$, and another selector "E_{anc}" with the same declaration. This statement might then be redundant as for E_{desc} the property might inherited from E_{anc}.

However we do not know if some document using this CSS file, presents a structure in which there is a certain element E_x placed in between E_{anc} and E_{desc}, and the property declarations for E_x alter the property inheritance among E_{anc} and E_{desc}. For example, a selector concerning only an attribute, such as "`[input]`", is free to be applied to any element on the document. Only certain CSS properties are inherited by default. Therefore, in our analyses, the amount of refactoring due to inheritance is a priori limited.

4. IMPLEMENTATION TECHNIQUES

4.1 Reasoning over selectors

Given two selectors, we intend to automatically check whether some containment or equivalence relation holds between them. For this purpose, we use the translation of CSS selectors into the tree logic described in [2] and obtain logical formulas. We then formulate containment as logical implication, and test the formula for satisfiability using the logical solver of [3]. For two selectors S_1 and S_2, Table 1 summarizes the tests performed, the four possible scenarios obtained according to the results, and the corresponding actions performed, as explained in section 3.1 and 3.2.

$S_1 \subseteq S_2$	$S_1 \supseteq S_2$	Relation	Action
0	0	None	None
0	1	$S_1 \supset S_2$	Refactoring 1, 2
1	0	$S_1 \subset S_2$	Refactoring 1, 2
1	1	$S_1 \Leftrightarrow S_2$	Rafactoring 3

Table 1: Actions associated with detected relations.

4.2 Processing on declaration blocks

Once a relation between selectors S_1 and S_2 has been found, for each of the properties declared under both rules, we determine whether it is necessary or not, depending on three aspects: the relation between selectors, the selector's specificity, and whether the properties share the same value; as illustrated on examples in sections 3.1 and 3.2.

In some cases, the deletion of unnecessary property declarations results in an empty rule. In this case, the whole rule is (safely) erased from the style sheet.

4.3 Optimization of elapsed time

In a style sheet with n rules, each rule can be tested against all rules but itself, adding up to a total of $n \times (n-1)$ possible tests. Given the diversity of elements in a HTML tree, tests concerning selector pairs such as "`body`" and "`p`" will not be uncommon. Only by adding a few basic pre checks, we will be able to determine the result of logical tests before actually processing it. We take advantage of two main observations for drastically reducing the number of pairwise tests:

1. *If two selectors point to elements with syntactically different names, they will never be contained in each other, in any of the two possible containment directions.*

2. *If a selector S_1 refers to one or more attributes that S_2 does not, S_1 will never contain S_2.*

4.4 Statistics tracking

The tool tracks some statistics about the analysis. First, while parsing it detects the total number of rules, the number of ignored ones, the number of possible tests, and the tests that were actually carried out. After all reasoning is done, the tool counts the number of relations between selectors, the modified rules, the number of deleted properties as well as the deleted bits. Finally, the time spent on each one of the analysis parts is also shown.

4.5 Room for improvements

Our prototype implements the aforementioned procedures for a significant CSS subset, sufficient for performing practical experiments with real-world style sheets. Our prototype can be improved in many respects, though. In particular, the library css-validator[1], that is used for parsing the CSS file and traversing the properties of the rules, could be improved. Some methods concerning comparisons of properties are not implemented, and thus some potential property deletions cannot be automatically carried out. It does not support browser specific properties yet.

Some CSS selectors' features are not supported yet, such as grouping, pseudo-classes, pseudo-elements, multiple class and id selectors and media queries. Consequently, the concerning selectors are ignored. With their implementation, additional refactoring might be performed.

5. EXPERIMENTAL RESULTS

In order to get a representative collection of results, three different groups have been defined. The first group involves CSS code provided by frameworks and CMS, and is represented by *Bootstrap*, *Joomla* and *JQuery*. The second group consists in style sheets from complex web applications, and features *Instagram*, *Twitter* and *The Times*. Finally, we have extracted CSS files from some random web sites of medium complexity, which are *ACM DL*, *DocEng*, and *Inovallée*. Table 2 provides detailed information about the corresponding CSS file sizes and complexity.

Name	# bytes	# rules
Bootstrap (*Framework's CSS*)	127343	805
Joomla *(Template Beez20's CSS)*	30158	325
JQuery *(Framework's CSS)*	32891	349
Instagram *(www.instagram.com)*	123815	791
The Times *(www.thetimes.co.uk)*	89362	469
Twitter *(www.twitter.com)*	245473	2402
ACM DL *(dl.acm.org)*	11151	97
DocEng *(www.doceng2014.org)*	204970	1571
Inovallée *(www.inovallee.com)*	29930	189

Table 2: Dataset for the experiments.

[1]see http://jigsaw.w3.org/css-validator/

5.1 CSS Size Reduction in Practice

After processing the aforementioned files, the tool has spotted on average 4.95%[2] of unnecessary property declarations, modifying 4.56% of the total rules. Of the relations found, 83.38% were containment ones, and the remaining 16.62% correspond to equivalence between selectors.

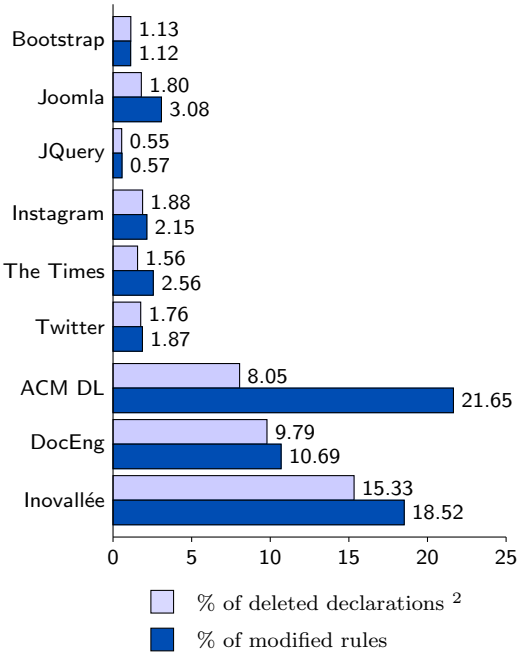

Figure 1: Refactoring performed.

It is clear that the style sheets from the first and second group present a low percentage of deleted property declarations (1.16% and 1.73% respectively). The same holds for the percentage of modified rules (1.59% for the first group and 2.19% for the second one). However, in the third group these numbers rise dramatically, reaching 11.05% of deleted declarations and 16.95% of modified rules. Although these latter sites might not have involved as much testing as the first ones, they are not amateur web sites either.

5.2 Performance of the tool

The time taken to analyze each file is shown in Figure 2. A 34.30% of the rules have been ignored due to unsupported selectors commented in Section 4.5, and an average of 99.88%[3] of the them has been discarded by the optimization mechanism described in Section 4.3. Each test between selectors requires a median of 156.71 ms, so without the optimization mechanism, each file analysis would have taken a median of 16.60 hours, in contraposition to the 78.16 seconds that were actually required, still guaranteeing the same results.

6. CONCLUSION

This paper presents a tool that automatically detects and removes unnecessary property declarations in CSS files, based on the analysis of semantical relations between selectors. We provide a first prototype implementation, with many perspectives for further development. Our method can constitute the core mechanism in several applications aimed to help developers code higher quality style sheets. A basic example could be a file processor where the user inputs a CSS file and gets as output an equivalent lighter file. The tool could also be integrated into more powerful components such as context features for web IDEs. In any scenario, the benefit of our tool is to conduct precise semantical analyses, that go far beyond the capabilities of purely syntactic analyses done by current CSS optimizers. Generating equivalent but simpler CSS files not only improves the time spent in loading and formatting a web page, but might also facilitate the debugging process of style sheets.

[2]calculated over the properties that the tool supports.
[3]calculated over considered tests, discarding ignored ones.

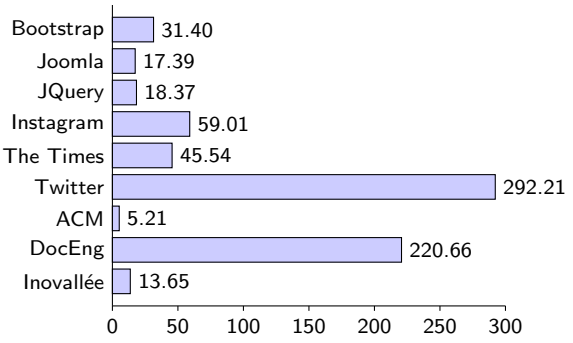

Figure 2: elapsed time (s)

Despite the large number of unsupported features, the results obtained in section 5 already validate our approach: we have been able to detect large numbers of unnecessary property declarations in non-amateur web pages; and we have also found mistakes in the style sheets of some of the most popular web sites. The number of safe modifications can easily grow as more components of CSS are supported and more features are implemented, such as property inheritance, translation of pseudo-classes into query languages, analysis of media queries, merging of equivalent selectors or containment involving grouped selectors.

A perspective for further work consists in extending the number and the precision of our analyses by supporting constraints on the document structure when they are available (as a DTD/Schema, or via another formalism such as [1]).

7. REFERENCES

[1] E. Benson and D. R. Karger. Cascading tree sheets and recombinant HTML: better encapsulation and retargeting of web content. In *WWW'13*, pages 107–118, 2013.

[2] P. Genevès, N. Layaïda, and V. Quint. On the analysis of cascading style sheets. In *WWW'12*, pages 809–818, 2012.

[3] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07*, pages 342–351, 2007.

[4] Google. Chrome developer tools, May 2014. https://developer.chrome.com/devtools/.

[5] H.-S. Liang, K.-H. Kuo, P.-W. Lee, Y.-C. Chan, Y.-C. Lin, and M. Y. Chen. SeeSS: Seeing what i broke – visualizing change impact of cascading style sheets (CSS). In *UIST'13*, pages 353–356, 2013.

[6] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *ICSE'12*, pages 408–418, 2012.

[7] Mozilla. Firebug, May 2014. https://getfirebug.com/.