HAL
open science

Reasoning with Style

Martí Bosch, Pierre Genevès, Nabil Layaïda

To cite this version:

Martí Bosch, Pierre Genevès, Nabil Layaïda. Reasoning with Style. International Joint Conference On Artificial Intelligence (IJCAI 2015), Jul 2015, Buenos Aires, Argentina. hal-01149248

HAL Id: hal-01149248

https://inria.hal.science/hal-01149248v1

Submitted on 6 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reasoning with Style

Martí Bosch[*] [†] [‡]
[*] Universitat Politècnica de Catalunya

Pierre Genevès[†] [‡]
[†] CNRS, LIG

Nabil Layaïda[†] [‡]
[‡] Inria

Abstract

The Cascading Style Sheets (CSS) language constitutes a key component of web applications. It offers a series of sophisticated features to stylize web pages. Its apparent simplicity and power are however counter-balanced by the difficulty of debugging and maintaining style sheets, tasks for which developers still lack appropriate tools. In particular, significant portions of CSS code become either useless or redundant, and tend to accumulate over time. The situation becomes even worse as more complex features are added to the CSS language (e.g. CSS3 powerful selectors). A direct consequence is a waste of CPU that is required to display web pages, as well as the significant amount of useless traffic at web scale.

Style sheets are designed to operate on a set of documents (possibly generated). However, existing techniques consist in syntax validators, optimizers and runtime debuggers that operate in one particular document instance. As such, they do not provide guarantees concerning all web pages in CSS refactoring, such as preservation of the formatting. This is partly because they are essentially syntactic and do not take advantage of CSS semantics to detect redundancies.

We propose a set of automated refactoring techniques aimed at removing redundant and inaccessible declarations and rules, without affecting the layout of any document to which the style sheet is applied. We implemented a prototype that has been extensively tested with popular web sites (such as Google Sites, CNN, Apple, etc.). We show that significant size reduction can be obtained while preserving the code readability and improving maintainability.

1 Introduction

Cascading Style Sheets (CSS) [Consortium, 2014; Lie, 2005] is a style sheet language used to format documents written in markup languages, and nowadays it is widely used to style web pages. To suit the current context, where so much attention is paid to the user experience in different kinds of devices, it includes a series of sophisticated features that offer a very wide range of possibilities to designers and developers.

Despite its widespread use and increasingly important role, CSS received very little attention from the research community [Quint and Vatton, 2007; Marden and Munson, 1999] with the notable exceptions of [Genevès *et al.*, 2012; Mesbah and Mirshokraie, 2012]. The simplicity of CSS syntax may be misleading as it hides the complexity of its most advanced aspects. For this reason, developers increasingly rely on frameworks that automatically generate CSS files. This often results in a code that is very hard to maintain. This also leads to redundant declarations and inaccessible selectors, that unnecessarily increase resources required to transfer the CSS files (and therefore web traffic at a global scale) and to process the page layout. Previous studies [Meyerovich and Bodik, 2010] showed that the visual layout of web pages consumes 40–70% of the processing time of the browser. Simplifying CSS code is essential in reducing this cost. Current syntax optimizers[1] perform only syntactic analyses, and are unaware of the CSS semantics. Therefore, they can only achieve a fraction of the potential optimizations.

The standard CSS development practice involves the use of empirical methods such as runtime debuggers. While these tools are useful in testing, they depend strongly on the document instance on which the style sheet is applied. However, as style sheets usually apply to a wider set of documents, modifications achieved on a particular instance might undesirably alter the presentation of other documents. On the other hand, CSS code very often comes from different sources, such as external files, the HTML's `style` element, or inline styles set directly as attributes on specific elements. This makes it hard to spot the origin of bugs, and incurs significant debugging costs. The tool we propose is intended to remove redundant code, clean and refactor CSS files, lessening the reliance on debuggers as well as reducing file's sizes. Our tool involves automated analysis of the style sheet semantics for performing size reductions which are not achievable by existing (syntactic) compressors.

Contributions and Outline We recall the main concepts of CSS in Section 2. We propose automated refactoring techniques in Section 3 that aim at removing redundant and inaccessible CSS declarations and rules, while preserving the layout of documents to which the style sheet is applied[2]. We implemented a prototype described in Section 4. We report on experimental results with CSS of popular web sites (such as Google Sites, CNN, Apple, etc.) in Section 5. We show

[1] For example: cleancss.com, codebeautifier.com, csslint.net, etc.

[2] Work partially supported by ANR TYPEX ANR-11-BS02-00.

that we obtain significant size reductions for these sites that represent a considerable fraction of web traffic.

2 The CSS Language

Cascading Style Sheets play nowadays a key role in the web infrastructure and in enhancing the user experience. Besides web developers, the simplicity of its syntax has attracted also graphical and web designers. CSS permits separating the content from presentation and a few rules are capable of producing impressively fancy presentations. A style sheet C can be seen as a sequence of rules, where each rule R consists of a selector S, and a set of declarations called declaration blocks. Selectors identify the set of elements that are styled by the declarations d_i. Each declaration d_i is a pair of a property p_i and its associated value v_i, that define how the elements selected by S will look like in the browser.

2.1 Selectors

CSS selectors decide which elements will be styled by the rules' declarations. The selectors' language is very expressive and permits matching elements based on the elements' structure and attributes, as well as grouping selectors that share the same declarations. Selectors Level 3 [Çelik *et al.*, 2011] also includes a series of advanced features that empower CSS' styling capabilities while inevitably making it more complex. An efficient use of selectors is one of the key aspects towards the conception of maintainable CSS code.

2.2 Specificity

When several rules apply to a given element, CSS prioritizes the declarations that have the `!important` specifier. In case of equality, CSS picks the declaration whose selector has a highest *specificity*. The *specificity* of a selector is represented by a four integer vector, whose components are, from most to less important, determined as: (1) 1 if the property is declared under the `style` attribute, 0 otherwise, (2) the number of references to the `id` attribute with the # symbol, (3) the number of class selectors, attribute selectors and pseudo-classes, and (4) the number of element type selectors and pseudo-elements. If the selectors have the same specificity, the latter rule in the syntactic order gets precedence.

2.3 Media Rules

A *media rule* starts by `@media` followed by a condition called *media query*. This defines a block in which we can add a set of CSS rules that only apply when the *media query* is satisfied. The *media queries* define the style sheet target according to *expressions* concerning devices' features, as for example the dimensions. Such features are key to define how web pages look like on mobile devices. To adjust more precisely the style sheet scope, *media queries* might use several logical operators to connect different *expressions*. It is important to be aware that media rules do not alter the specificity.

3 CSS Refactoring Techniques

In this section, we introduce a series of refactoring techniques capable of identifying and deleting unaccessible declarations and combine rules in a way such that the same rendering semantics is preserved with lighter CSS files. These techniques do not require any information other than the style sheet itself, regardless of which instance is under consideration. For this purpose, our methods rely on the semantics of the CSS components described previously. In particular, our method aim at analysing and leveraging the fact that in CSS, rules use selectors that match the set of elements that are styled by the rule's declarations. Our main method consists in the detection of semantic relations between CSS selectors. When some of these relations are detected, further analysis of some CSS aspects might reveal that some declarations are in fact unnecessary and can be deleted.

3.1 Deleting redundant declarations

In the context of this paper, we will state a CSS declaration as *redundant* if (1) it is *always masked* by another declaration, or (2) it is *verbose* as the styling that it provides is already provided by some other declaration.

Masked declarations

A declaration d_a is *always* inactive iff it is *masked* by some other declaration d_b concerning the same CSS property. In order to mask d_a, d_b must apply to at least the same set of elements as d_a, and this is why relations between the selectors under which d_a and d_b are stated become crucial: selectors must either hold an equivalence or a containment relation.

Listing 1: Example of *masked* declaration

```
1 li.foo { text-indent: none;
2          color: blue;
3          font-weight: bold }
4 li[class='foo'] { text-indent: 10px;
5                   color: blue }
```

In Listing 1 we have two rules with equivalent selectors: "`li.foo`" and "`li[class='foo']`" both select an `li` element with the `class` attribute set to "`foo`". They just use different syntax to achieve the same semantics. Any element matched by the first rule will also be matched by the second one and viceversa, and as both rules have declarations concerning the properties `color` and `text-indent`, CSS specificity will decide which values will apply. In Listing 1, both selectors have the same specificity, so the declarations under the latter one "`li[class='foo']`" will have preference. Consequently, for these conflicting declarations, the values from "`li[class='foo']`" will apply, and the ones from "`li.foo`" will be *always* inactive (masked). By deleting those declarations we achieve a lighter code with the same rendering semantics: Listing 2.

Listing 2: Code after deleting the *masked* declarations

```
1 li.foo { font-weight: bold }
2 li[class='foo'] { text-indent: 10px;
3                   color: blue }
```

Consider we replace "`li[class='foo']`" by "`li[class]`" in Listing 1. The selector "`li[class]`" matches any `li` element with the `class` attribute set (to any value). Notice then that "`li.foo`" $\subset$ "`li[class]`",

given that any `li` element with the `class` attribute set to "`foo`" does indeed have the `class` attribute set, but not all the `li` elements with the `class` attribute defined need to have it set to the value "`foo`". Observe then that the set of elements pointed by "`li.foo`" will also be pointed by "`li[class]`", so the declarations concerning the properties `color` and `text-indent` set under "`li.foo`" will be completely masked by those set under "`li[class]`", as "`li[class]`" gets preference as it has the same specificity but is stated after "`li.foo`".

Let us note by $d_{b \wedge p}$ the set of declarations concerning a CSS property that is stated under both selectors S_b and S_p. When there are two rules with selectors such that $S_b \subseteq S_p$ and S_p has preference over S_b, the deletion of declarations that are *masked* is carried automatically by Procedure 1:

Procedure 1: $S_b \subseteq S_p$ with S_p preferred over S_b

foreach d_i *in* $d_{b \wedge p}$ **do**
  delete d_i from S_b
end

Verbose declarations

A CSS declaration might be *active* in some occasions and yet not provide any additional styling. In the context of this paper, they will be noted as *verbose* declarations. To understand how we might find them, consider Listing 3.

Listing 3: Example of *verbose* declarations

```
1 div div > a { font-size: 14px;
2               text-decoration: none }
3 div a { font-size: 14px;
4         text-decoration: underline }
```

There are two rules whose selectors hold a containment relation "`div div > a`" $\subset$ "`div a`". Note that any `a` element with a `div` father and another `div` ancestor also matches the pattern of a `a` element with a `div` ancestor. In this case "`div div > a`" has highest specificity than "`div a`", so when their declarations conflict, the ones stated under "`div div > a`" dominate. Consequently for an element selected by the subset (and implicitly by the superset as well), the value for `text-decoration` will be `none`. Nevertheless, for the same element, `font-size` is set to `14px` by both selectors, so although the value is pulled from the subset, it would always be pulled from the superset as well, and thus the declaration "`font-size:14px`" under "`div div > a`" is *verbose* and can be deleted, yielding Listing 4.

Listing 4: Code after deleting the *verbose* declarations

```
1 div div > a { text-decoration: none }
2 div a { font-size: 14px;
3         text-decoration: underline }
```

This refactoring can be automatically performed by Procedure 2. Such procedure can only be applied to a subset in *strict* containment, as in the case of equivalence Procedure 1 would apply due to the symmetric nature of the relation.

Procedure 2: $S_b \subset S_p$ with S_b preferred over S_p

foreach d_i *in* $d_{b \wedge p}$ **do**
  if d_i *sets the same value under both* S_b *and* S_p **then**
    delete d_i from S_b
  end
end

3.2 Relations and Media Rules

Media queries may also cause certain inconsistencies. For example, observe in Listing 5 that a device that satisfies the first media query will always satisfy the second one as well. And considering that media rules do not alter selector's specificity, in this case "`color: red`" will never be applied. Note that, if we reverse the order of the media rules, "`color: blue`" will be active for devices wider than `800px` but smaller than `1000px`, so the color of the `div` elements would be blue. One can reasonably guess that this was the behaviour that the developer intended.

Listing 5: Relations and Media Rules

```
1 @media (min-width: 1000px) {
2   div { color: red } }
3 @media (min-width: 800px) {
4   div { color: blue } }
```

These inconsistencies can be detected too. Given two media rules, they will either hold: (a) no relation (which will be the case most of the time), (b) a *containment* relation, like in Listing 5 or (c) an equivalence relation. In order to perform the refactoring procedures proposed in Section 3.1 in presence of media rules, we have to see how selectors' relations change according to their respective media context, which can happen in two ways:

1. Destruction of relations consider a style sheet with S_a = "`div p`" under a media query mQ_a = "`all`" and S_b = "`p`" under mQ_b = "`tv`". Note that "`div p`" $\subset$ "`p`", but "`div p`" applies in `all` media contexts whereas "`p`" is only active for `tv` media types. So whenever we are not under a `tv` device, "`div p`" might point to elements that "`p`" would not, and thus it *breaks* the nature of the containment relation, and prevents our tool from performing the refactorings proposed in Section 3.1, which become no longer safe to achieve.

2. Equivalence transformed into containment consider again Listing 5. We have two identical selectors S_a, S_b = "`div`", in two different media contexts, mQ_a = "`(min-width: 1000px)`" and mQ_b = "`(min-widith: 800px)`", being true that $mQ_a \subset mQ_b$. Under these circumstances, any device satisfying mQ_a will also satisfy mQ_b, but not the other way around. So whenever S_a selects all the `div` elements, S_b will do too, but not conversely. In short, an equivalence relation is turned into containment.

After both cases are considered, the refactoring procedures from Section 3.1 can be performed in presence of media rules.

3.3 Deleting Empty Rules

The procedures above are meant to delete declarations from CSS rules. Applying such procedures might result in an *empty* rule containing a selector with no declarations. This rule does select a set of elements but does not define any styling for them. Consequently, it is safely deleted.

3.4 Merging Rules with Equivalent Selectors

Given two equivalent selectors S_a and S_b, merging their respective rules R_a and R_b can reduce the style sheet's size. By merging R_a and R_b we mean the following: as S_a and S_b affect the same set of elements, we group both rules' declarations. We keep S_a and move all the declarations from R_b to R_a. If any declaration concerns the same property with different values set under R_a and R_b, Section 3.1 already takes care of it. To exemplify the procedure, consider Listing 6.

Listing 6: Merging rules with equivalent selectors

```
1 a#nav { padding: 0px;
2         height: 20px }
3 a[id='nav'] { margin: 5px }
4 p a[id='nav'] { margin: 10px }
```

We have two equivalent selectors "`a#nav`" $\Leftrightarrow$ "`a[id='nav']`", so we keep the rule with "`a#nav`", and move "`margin: 5px`" from "`a[id='nav']`" to "`a#nav`". Instead of 2 rules with 2 and 1 declarations each, after this refactoring we have 1 rule with 3 declarations.

However, we cannot perform this refactoring as *it does not preserve the style sheet's semantics*. The reason is because selectors confer its specificity to the declarations that they encapsulate. For example, in Listing 6, when we move "`margin: 5px`" from "`a[id='nav']`" to "`a#nav`", for an element pointed by "`p a[id='nav']`", the declaration "`margin: 10px`" will always be inactive as "`p a[id='nav']`" $\subset$ "`a#nav`", and "`a#nav`" will confer its *higher* specificity to its declaration "`margin: 5px`", and so it would override "`margin: 10px`". If we do not refactor Listing 6, "`margin: 5px`" has the specificity from its selector "`a[id='nav']`", which is lower than the one from "`p a[id='nav']`". Therefore, for the set of elements matched by "`p a[id='nav']`", `margin` will be `10px`.

Applicability of the Procedure To safely merge a pair of rules R_a and R_b with equivalent selectors $S_a \Leftrightarrow S_b$, we need to guarantee that there is no selector S_c or S_d in the style sheet such that: (i) $S_c \subset S_a$ with S_c preferred over S_a, and S_b preferred over S_c or (ii) $S_d \supset S_a$ with S_d preferred over S_a, and S_b preferred over S_d. Note that as $S_a \Leftrightarrow S_b$, the relation $S_c \subset S_a$ already implies that $S_c \subset S_b$, and the same applies to S_d. More generally, a pair of rules with equivalent selectors $S_a \Leftrightarrow S_b$ might be merged if *all* the selectors that hold a *containment* relationship with them, are either *more specific* than both S_a and S_b, or are *less specific* than both S_a and S_b. Observe that in Listing 6, the subset "`p a[id='nav']`" is *more specific* than "`a[id='nav']`" but *less specific* than "`a#nav`", and that is why the rules cannot be merged.

However, if a rule R_x with a selector S_x fulfilling the conditions of (i) or (ii) was found, the rules R_a and R_b might still be merged only if the declarations we move among R_a and R_b *do not concern any of the properties* declared under R_x. In Listing 6, we can safely move "`padding: 0px`" and "`height: 20px`" from the rule with the selector "`a#nav`" to the one with "`a[id='nav']`", and delete the resulting empty rule "`a#nav { }`". So after merging "`a#nav`" and "`a[id='nav']` while preserving the semantics, we obtain Listing 7. In realistic style sheets, a much larger set of rules need to be considered in order to determine if this refactoring is valid or not.

Listing 7: After merging rules the right way

```
1 a[id='nav'] { margin: 5px;
2               padding: 0px;
3               height: 20px }
4 p a[id='nav'] { margin: 10px }
```

4 Implementation techniques

4.1 Modeling & Analysis of Single Selectors

To detect relations between selectors, we first translate selectors in a logic: each selector S_i is associated with a logical formula $\mathtt{F}(S_i)$, and then we issue external calls to a logical satisfiability solver such as the one of [Genevès *et al.*, 2007; Genevès *et al.*, 2015] to check for the existence of relations between the formulas. For instance, the validity of a containment relation $S_i \subseteq S_j$ is checked by testing for the unsatisfiability of the negation of the logical implication $\mathtt{F}(S_i) \implies \mathtt{F}(S_j)$.

4.2 Grouped Selectors

A grouped selector is a list of $n > 1$ single selectors, separated by a comma character. The translation of a grouped selector S_g intro tree logic requires the translation of each one of the n single selectors. The comma character plays the role of a logical "`or`" between the different single selectors that compose the group, so the logical disjunction operator `|` will be used to connect the single selectors' translations, capturing this way the semantics of the grouped selector. Given a grouped selector S_g, composed by n single selectors $s_{g,i}$, we have: $\mathtt{F}(S_g) = \mathtt{F}(s_{g,1})$ `|` $\mathtt{F}(s_{g,2})$ `|` ... `|` $\mathtt{F}(s_{g,n})$

Specificity and grouped selectors Given a pair of single selectors, to determine which one gets precedence, we calculate their *specificity* vectors, and compare their components. Nevertheless, *specificity cannot be determined for grouped selectors*. Instead, our tool includes an advanced mechanism to determine, if possible, which selector will get precedence when grouped selectors are present. Sometimes this cannot be determined, and our tool does not refactor the code as it might not preserve its semantics.

4.3 Traversal of Selectors

Given a style sheet with N rules, there will be N selectors S_i where $i = 1, 2, \ldots, N$. To detect all the possible relations between selectors, each S_i has to be tested for logical inclusion against the remaining $N - 1$ selectors. This adds up to a total of $N \times (N - 1)$ tests, which for style sheets with more than 1000 rules results in *millions* of tests. Since logical tests

are the heaviest computation of our tool, several mechanisms can be used to avoid such explosion:

1. *If two selectors point to elements with different names, they will always represent disjoint sets*

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

3. *Given a pair of selectors such that $S_1 \Leftrightarrow S_2$, and a third selector S_3, if (a) $S_3 \subset S_1$, (b) $S_3 \supset S_1$ or (c) $S_3 \Leftrightarrow S_1$, then (a) $S_3 \subset S_2$, (b) $S_3 \supset S_2$ or (c) $S_3 \Leftrightarrow S_2$ due to the transitivity of equivalence and containment binary relations*

5 Experimental Results

We now report on the experimental evaluation of our refactoring methods with real-world style sheets used in some of the most popular web sites.

5.1 Analysis of Single CSS Files

We have extracted in Table 1 the largest CSS file used in 20 different websites corresponding to various web applications types and having different complexity levels. For the sake of brevity, an identifier (integer) is used to identify each of them. The number of CSS rules is shown in this table. This number gives a clear idea of the style sheet's size and constitutes a relevant metric for the techniques described in this paper. The file sizes range from 10 to 320 Kilobytes.

ID	Web	# CSS Rules	ID	Web	# CSS Rules
1	ACM Digital Lib.	102	11	Google Sites[4]	1676
2	Aerolineas Argentinas	1284	12	IJCAI-15	384
3	Apple	784	13	Lamborghini	1472
4	Argentina Travel	651	14	Microsoft	702
5	Clarin	1188	15	Opera	708
6	CNN	2738	16	PayPal	1089
7	Coursera	3690	17	Salesforce	1158
8	Ebay	1573	18	Shell	1111
9	Facebook	2757	19	Univ. of Cambridge	845
10	Foundation[3]	800	20	YouTube	1841

Table 1: Dataset for the experiments

In the remaining part of this Section, whenever global averages of percentages are calculated, we take the total deletions in all files and compare it to the total files' size. Our tool prototype does not yet support all CSS3 selectors[5], so the percentages are determined *relative* to the part of the style sheet that our prototype supports.

For each file, the size reduction achieved by applying all the refactoring techniques from Section 3 is shown in Figure 1. On average, the style sheets sizes have been reduced a 7.75 %. A 7.79 % of the declarations have been *safely* deleted, modifying a 13.66 % of the rules and leaving a 4.71 % of the total style sheet's rules empty.

Deleting redundant declarations with the procedures from Section 3.1 contributes to a 60.22 % of the total size reduction, whereas deleting the rules that became *empty* (as proposed in Section 3.3), contributes to a 31.70 %. Merging rules

[3]ZURB Foundation framework's default template

[4]Google Sites' default template

[5]As of February 2015, 69.46 % of selectors in the dataset of Table 1 are supported.

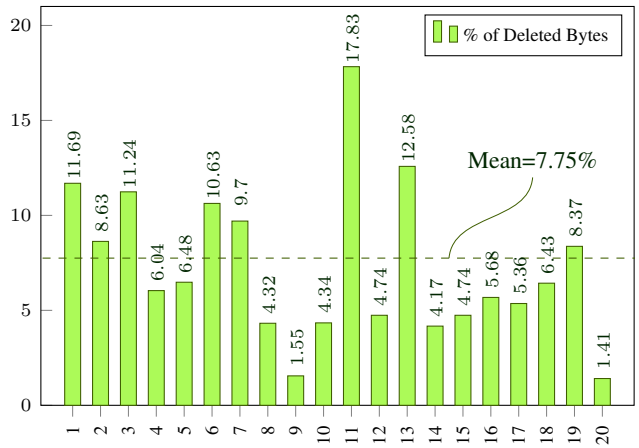

Figure 1: % of deleted bytes for each data set entry

as explained in Section 3.4 only contributes by an 8.08 % size reduction (in bytes). Media Rules are considered in this analysis, but they are not a procedure itself but transform selectors' relations, as explained in Section 3.2.

Besides the averages, the percentages of file size reduction in bytes vary dramatically among the style sheets, and so does the contribution of the different refactorings proposed in Section 3. This shows that the inconsistencies found on the style sheets depend mostly on the developers, which suffer from the lack of tools like the one that we propose.

5.2 Style Sheets with Multiple Files

Modern web applications often use several CSS files to define their presentation. Each file might correspond to the styling of distinct sections of the application, different parts of the same page, or some groupings which resembles libraries.

Regardless of the specific instances of the HTML involved, our tool can achieve the refactoring procedures across several CSS files. The information needed for the analysis is extracted from HTML documents: it consists in exploring the `!DOCTYPE` declaration, and the `link` elements that are under the `head` element. For each `link` element with the attribute `rel="stylesheet"`, we retrieve the CSS file referenced by the `href` attribute, and if the `media` attribute is set, we extract its value. Such a value corresponds to a media query setting the style sheet's media type. If the `media` attribute is not set, then the style sheet's media type takes the default value as described in the specification: `all` in HTML5 or `screen` in HTML 4 and XHTML 1, which is why we need to extract `!DOCTYPE` specification.

To analyze style sheets with several files, we have retained the web sites of our dataset whose main HTML page has more than one CSS file referenced in `link` elements. This new dataset is shown in Table 2, as well as the extra information extracted for this analysis and the total number of rules.

One of the many CSS subtleties is that declarations from one file might become redundant because of declarations stated in some other files. To see how this might affect the results of our refactoring procedures, in this section, we perform two experiments for each of the items listed in Table 2. The first experiment consists in analyzing all of the item's CSS files separately, as if they were used in isolation and

ID	Web	DocType	# CSS Files	Default Media	# CSS Rules
9	Facebook	HTML5	5	`all`	3543
11	Google Sites[4]	XHTML 1	6	`screen`	2046
12	IJCAI-15	XHTML 1	14	`screen`	1170
15	Opera	HTML5	2	`all`	890
16	PayPal	HTML5	3	`all`	1186
18	Shell	HTML5	6	`all`	1551
20	YouTube	HTML5	4	`all`	3615

Table 2: Dataset for the experiments.

without interaction between them. For the second experiment, we use the information extracted from the `link` elements to emulate the global compounded style sheet that the browser will effectively use, and we then apply our refactoring over it. The refactoring that our tool performs in this second experiment does not alter the presentation of any HTML instance, *as long as* the instance: (1) has the same `!DOCTYPE` declaration as the site's default (listed in Table 2), and (2) includes *at least the same* references to external CSS files (the `link` elements under `head`). If (1) or (2) are not satisfied, then *our tool does not guarantee the semantic preservation of the presentation*.

The average time spent for the overall static analysis of a dataset entry is 91 s, with an average time of 49 ms spent in each logical solver test.

For each item in Table 2, Figure 2 shows the percentages of deleted bytes for the two experiments. We can observe that for every item there exists some *interaction* between the different files, since more bytes are *always* deleted in the second experiment. Overall, Figure 2 shows that, in the setting of style sheets using several files, rules from different files tend to collide, and in some cases this results in many more inaccessible and redundant declarations.

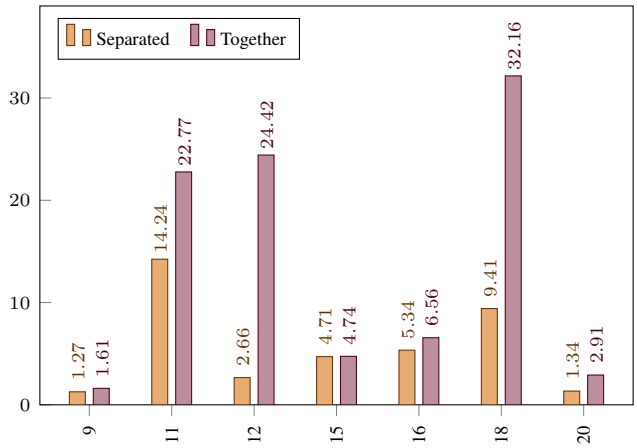

Figure 2: % of deleted bytes on multiple-file style sheets.

6 Related Work

Little research effort has been dedicated to study the CSS language and fewer have been dedicated to the quality of widely deployed style sheets. The work found in [Liang *et al.*, 2013] explores a visual approach to track the effect of source code modifications. This approach can be complementary to existing runtime debuggers available in current browsers [Mozilla, 2014; Google, 2014; Opera, 2014]. [Keller and Nussbaumer, 2009; 2010; Benson, 2013; Sinha and Karim, 2013] focus on the reusability of CSS for different platforms, whereas [Benson and Karger, 2013] introduces a formalism to better separate CSS structural and styling components.

The work found in [Genevès *et al.*, 2012] does not consider CSS refactoring nor their compression, but it introduces a translation of CSS selectors into a tree logic for the purpose of detecting bugs. Our paper relies on an generalisation and extension of such a translation for the part concerned with the detection of relations between selectors.

Closer to our work we find [Mesbah and Mirshokraie, 2012] that proposed an analysis whose purpose is to dynamically detect unused declarations. The most closely related research work can be found in [Mazinanian *et al.*, 2014]. The authors present layout-preserving refactoring techniques for CSS based on the detection of rules with very similar declaration blocks. Selectors are grouped and shorthands are used for recombining those rules. Their tool is tested with a large dataset and also obtains interesting size reductions. A fundamental difference with our approach is that their analysis is incomplete and conducted at runtime on a particular instance. In contrast, our static analysis technique needs to be performed only once and the size reduction is observed every time a page is fetched by a different user. Furthermore, the refactoring achieved by [Mazinanian *et al.*, 2014] can be used alongside those obtained in the present paper, as our tool is capable of actually *deleting* declarations rather than recombining or compressing them. One promising research work would consist in investigating a technique combining both static and dynamic analyses.

7 Conclusion and Perspectives

We present techniques and a tool capable of automatically refactoring CSS files with the aim of reducing the sizes of style sheets, while preserving their rendering semantics. In contrast with the existing tools which are mostly dynamic and operate with a particular document instance, our techniques are based on the static analysis of semantic relations between CSS selectors and media queries. Our refactoring applies on a given style sheet, independently of any particular document instance, and reduces the style sheet size once for all. Our technique can be used in combination with existing syntactic optimisers since it performs size reductions that the latter cannot do. Experimental results show that our approach is capable of reducing significantly the size of CSS files of some of the most popular and sophisticated web sites. For the CSS files tested separately, our preliminary prototype has achieved an average size reduction of 7.75 %, with a maximum of 17.83 %. Furthermore, we showed that, with very little information about the documents that use a style sheet, the size reduction can increase up to 30 %.

One promising perspective for further work consists in extending the analyses by taking into account constraints, such as DTDs or schemas (like those defined in e.g. the XHTML Mobile Profile for mobile phones). The knowledge of additional structural constraints would certainly be beneficial in detecting even more refactoring opportunities.

References

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

[Benson, 2013] Edward Benson. Mockup driven web development. In *Proceedings of the 22nd International Conference on World Wide Web Companion*, WWW '13 Companion, pages 337–342, 2013.

[Çelik *et al.*, 2011] Tantek Çelik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, and John Williams. Selectors level 3. W3C recommendation, World Wide Web Consortium, September 2011.

[Consortium, 2014] World Wide Web Consortium. CSS specifications, November 2014. http://www.w3.org/Style/CSS/current-work.

[Genevès *et al.*, 2007] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07*, pages 342–351, 2007.

[Genevès *et al.*, 2012] Pierre Genevès, Nabil Layaïda, and Vincent Quint. On the analysis of cascading style sheets. In *WWW'12*, pages 809–818, 2012.

[Genevès *et al.*, 2015] Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. Efficiently Deciding μ-Calculus with Converse over Finite Trees. *ACM Transactions on Computational Logic*, 16(2), 2015.

[Google, 2014] Google. Chrome Developer Tools, November 2014. https://developer.chrome.com/devtools/.

[Keller and Nussbaumer, 2009] Matthias Keller and Martin Nussbaumer. Cascading style sheets: a novel approach towards productive styling with today's standards. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 1161–1162, 2009.

[Keller and Nussbaumer, 2010] Matthias Keller and Martin Nussbaumer. CSS code quality: A metric for abstractness. In *Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 116–121, October 2010.

[Liang *et al.*, 2013] Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. SeeSS: Seeing what i broke – visualizing change impact of cascading style sheets (CSS). In *UIST'13*, pages 353–356, 2013.

[Lie, 2005] Hkon Wium Lie. *Cascading style sheets*. Phd thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2005.

[Marden and Munson, 1999] Philip M. Marden and Ethan V. Munson. Today's style sheet standards: the great vision blinded. *Computer*, 32(11):123–125, nov 1999.

[Mazinanian *et al.*, 2014] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, page 11 pages, 2014.

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

[Meyerovich and Bodik, 2010] Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 711–720, 2010.

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

[Opera, 2014] Opera. Opera Dragonfly, November 2014. http://www.opera.com/dragonfly/.

[Quint and Vatton, 2007] Vincent Quint and Irène Vatton. Editing with style. In *Proceedings of the 2007 ACM symposium on Document engineering*, DocEng '07, pages 151–160, 2007.

[Sinha and Karim, 2013] Nishant Sinha and Rezwana Karim. Compiling mockups to flexible uis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 312–322, 2013.