Detecting Redundant CSS Rules in HTML5 Applications
A Tree Rewriting Approach

Matthew Hague
Royal Holloway University of London

Anthony Widjaja Lin
Yale-NUS College

Luke Ong
University of Oxford

HTML5 applications normally have a large set of CSS (Cascading Style Sheets) rules for data display. Each CSS rule consists of a node selector (given in an XPath-like query language) and a declaration block (assigning values to selected nodes’ display attributes). As web applications evolve, maintaining CSS files can easily become problematic. Some CSS rules will be replaced by new ones, but these obsolete (hence redundant) CSS rules often remain in the applications. Not only does this “bloat” the applications, but it also significantly increases web browsers’ processing time. Most works on detecting redundant CSS rules in HTML5 applications do not consider the dynamic behaviors of HTML5 (specified in JavaScript); in fact, the only proposed method that takes these into account is dynamic analysis (a.k.a. testing), which cannot soundly prove redundancy of CSS rules.

In this paper, we introduce an abstraction of HTML5 applications based on monotonic tree-rewriting and study its “redundancy problem”. We establish the precise complexity of the problem and various subproblems of practical importance (ranging from \(P\) to \(\text{EXP}\)). In particular, our algorithm relies on an efficient reduction to an analysis of symbolic pushdown systems (for which highly optimised solvers are available), which yields a fast method for checking redundancy in practice. We implemented our algorithm and demonstrated its efficacy in detecting redundant CSS rules in HTML5 applications.

Keywords HTML5, jQuery, CSS, redundancy analysis, static analysis, tree-rewriting, symbolic pushdown systems

1. Introduction

HTML5 is the latest revision of the HTML standard of the World Wide Web Consortium (W3C), which has become a standard markup language of the Internet. HTML5 provides a uniform framework for designing a web application: (1) data content is given as a standard HTML tree, (2) rules for data display are given in Cascading Style Sheets (CSS), and (3) dynamic behaviors are specified through JavaScript.

An HTML5 application normally contains a large set of CSS rules for data display, each consisting of a (node) selector given in an XPath-like query language and a declaration block which assigns values to selected nodes’ display attributes. As a web application evolves, some rules will be replaced by new rules. Developers often forget to remove these obsolete (hence redundant) rules, which “bloat” the application. A recent case study \cite{36} shows that in several industrial web applications, up to 60% of the CSS rules are redundant. These bloated applications are not only harder to maintain, but they also significantly increase web browsers’ processing time. In fact, a recent study \cite{37} reports that when web browsers are loading popular pages at least 34% of the CPU time is spent on CSS selectors (18%), layout (4%), and rendering (12%). [These numbers are calculated without even including the extra 31% uncategorised operations of the total CPU time, which could include operations from these three categories.] This suggests the importance of detecting and removing redundant CSS rules in an HTML5 application.

Indeed, a sound and automatic redundancy checker would allow bloated CSS stylesheets to be streamlined during development, and generic stylesheets to be minimised before deployment.

There has been a lot of works on optimising CSS (e.g. \cite{16, 22, 35–37}), which include merging “duplicated” CSS rules, refactoring CSS declaration blocks, and simplifying CSS selectors, to name a few. However, most of these works examine the set of CSS rules in isolation. In fact, the only available method that takes into account the dynamic nature of HTML5 introduced by JavaScript uses simple dynamic analysis (a.k.a. testing), which cannot soundly prove redundancy of CSS rules since the technique cannot in gen-
eral test all possible behaviors of the HTML5 application. For example, from the benchmarks of [36] there are some non-redundant CSS rules that their tool CIIA falsely identifies as redundant, e.g., due to browser-specific behavior under certain HTML5 tags like `<input/>` (see Section 6 for more details). Obviously, removing such rules can distort the presentation of HTML5 applications, which is undesirable.

A different approach to identifying redundant CSS rules is static analysis of HTML5. Since JavaScript is a Turing-complete programming language, the best one can hope for is approximating the behaviors of HTML5 applications. For the purpose of soundly identifying redundant CSS rules, we need a technique for computing a symbolic representation of an overapproximation of the set of all reachable HTML trees that is sufficiently precise for real-world applications. Existing static analysers do not address the problem of identifying redundant CSS rules. This is partly because there is currently no clean model that captures common dynamics of the HTML (DOM) tree caused by the JavaScript component of an HTML5 application and, at the same time, is amenable to algorithmic analysis. Such a model is not only important from a theoretical viewpoint, but it can also serve as a useful intermediate language for the analysis of HTML5 applications which among others can be used to identify redundant CSS rules.

The tree-rewriting paradigm — which is commonly used in database theory (e.g. [12, 13, 17, 20, 21, 32]) and infinite-state verification (e.g. [10, 23, 28–31]) — offers a clean theoretical framework for modelling the dynamics of tree updates and usually lends itself to fully-algorithmic analysis. This makes tree-rewriting a suitable framework in which to model the dynamics of tree updates commonly performed by HTML5 applications. Surveying real-world HTML5 applications (including [41] and real-world examples from the benchmark in [36]), we were surprised to learn that one-step tree updates used in these applications are extremely simple, despite the complexity of the JavaScript code from the point of view of static analysers. That said, we found that these updates are not restricted to modifying only certain regions of the HTML tree. As a result, models such as ground tree rewrite systems [30] and their extensions [23, 28, 31] (where only the bottom part of the tree may be modified) are not appropriate. However, systems with rules that may rewrite nodes in any region of a tree are problematic since they render the simplest problem of reachability undecidable. Recently, owing to the study of active XML, some restrictions that admit decidability of verification (e.g. [12, 13, 20, 21]) have been obtained. Despite this, these models have very high complexity (ranging from double exponential time to nonelementary), which makes practical implementation difficult.

**Contributions.** The main contribution of the paper is to give a simple and clean tree-rewriting model which strikes a good balance between: (1) expressivity in capturing the dynamics of tree updates commonly performed in HTML5 applications (esp. insofar as detecting redundant CSS rules is concerned), and (2) decidability and complexity of rule redundancy analysis (i.e. whether a given rewrite rule can ever be fired in a reachable tree). We show that the complexity of the problem is EXP-complete, though under various practical restrictions the complexity becomes PSPACE or even P. This is substantially better than the complexity of the more powerful tree rewriting models studied in the context of active XML, which is at least double-exponential time. Moreover, our algorithm relies on an efficient reduction to a reachability analysis in symbolic pushdown systems for which highly optimised solvers (e.g. Bebop [15], Getafix [25], and Moped [38]) are available.

We have implemented our reduction, together with a proof-of-concept translation tool from HTML5 to our tree rewriting model. Our translation specifically focuses on modelling standard features of jQuery [24] — a simple JavaScript library that makes HTML document traversal, manipulation, event handling, and animation easy from a web application developer’s viewpoint — since its use is so widespread in HTML5 applications nowadays (e.g., used in more than half of the top hundred thousand websites [3]) some authors [26] have advocated a study of jQuery as a language in its own right. Our experiments demonstrate the efficacy of our techniques in detecting redundant CSS rules in HTML5 applications. Furthermore, unlike dynamic analysis, our techniques will not falsely report CSS rules that may be invoked as redundant (at least within the fragment of HTML5 applications that our prototypical implementation can handle). We demonstrate this on specific examples from the benchmarks of [36].

**Organisation** We give a quick overview of HTML5 applications via a simple example in Section 2. We then introduce our tree rewriting model in Section 3. Since the general model is undecidable, we introduce a monotonic abstraction in Section 4. We provide an efficient reduction from an analysis of the monotonic abstraction to symbolic pushdown systems in Section 5. Experiments are reported in Section 6. We conclude with future work in Section 7. Missing proofs and further technical details can be found in the full version [2].

## 2. HTML5: a quick overview

In this section, we provide a brief overview of a simple HTML application. We assume basic familiarity with the static elements of HTML5, i.e., HTML documents (a.k.a. HTML DOM document objects) and CSS rules (e.g. see [6]). We will discuss their formal models in Section 3. Our example (also available at the URL [7]) is a small modification of an example taken from an online tutorial [4], which is given

---

1 Handling JavaScript in its full generality is a difficult engineering problem, which is beyond the scope of this paper.
in Figure 1. To better understand the application, we suggest the reader opens it with a web browser and interacts with it.

In this example the page displays a list of input text boxes contained in a div with class input_wrap. The user can add more input boxes by clicking the “add field” button, and can remove a text box by clicking its neighbouring “remove” button. The script, however, imposes a limit (i.e. 10) on the number of text boxes that can be added. If the user attempts to add another text box when this limit is reached, the div with ID limit displays the text “Limits reached” in red.

This dynamic behavior is specified within the second <script/> tag (the first simply loads the jQuery library). To understand the script, we will provide a quick overview of jQuery calls (see [24] for more detail). A simple jQuery call may take the form

$(selector).action(...);

where ‘$’ denotes that it is a jQuery call, selector is a CSS selector, and action is a rule for modifying the subtree rooted at this node. For example, in Figure 1, we have

$("#limit").addClass("warn");

The CSS selector #limit identifies the unique node in the tree with ID limit, while the addClass() call adds the class warn to this node. The CSS rule

.warn { color: red }

appearing in the head of the document will now match the node, and thus its contents will be displayed in red.

Another simple example of a jQuery call in Figure 1 is

$(".warn").removeClass("warn");

in Figure 1. This specifies that the function in ‘...’ should fire when a node with class button is clicked. Similarly, $(".input_wrap").on('click', '.remove', function(e){
	$(this).parent('div').remove();
	$("#counter").html(x);
	$("#limit").removeClass("warn");
});

adds a click listener to any node within the input_wrap div that has the class remove.

In general, jQuery calls might form chains. E.g.

$(this).parent('div').remove();

In this line, the call $(this) selects the node which has been clicked. The call to parent() and then remove() moves one step up the tree and if it finds a div element, removes the entire subtree (of the form <div><input/><a/></div>) from the document.

In addition to the action remove() which erases an entire subtree from the document, Figure 1 also contains other actions that potentially modify the shape of the HTML tree. The first such action is append(string1), which simply appends string1 at the end of the string inside the selected node tag. Of course, string1 might represent an HTML tree; in our example, it is a tree with three nodes. So, in effect append() adds this tree as the right-most child of the selected node. The second such action is html(string1), which first erases the string inside the selected node tag and then appends it with string1. In effect, this erases all the descendents of the selected node and adds a forest represented by string1.

An example where the CSS rule in Figure 1 becomes redundant is when the limit on the number of boxes is removed from the application (in effect, removing x), but the CSS is

Unlike node IDs, a single class might be associated with multiple nodes.
not updated to reflect the change (e.g. see [8]). In general, CSS selectors are non-trivial. For example

```
.a .b .c { color: red }
```

matches all nodes with class c and some ancestor containing both classes a and b. Thus, detecting redundant CSS rules requires a good knowledge of the kind of trees constructed by the application. In practice, redundant CSS rules easily arise when one modifies a sufficiently complex HTML5 application (the size of the top 1000 websites has recently exceeded 1600K Bytes [1]). Some popular web pages are known to have up to 60% redundant CSS rules, as suggested by recent case studies [36].

3. A tree-rewriting approach

In this section, we present our tree-rewriting model. Our design philosophy is to put a special emphasis on model simplicity and fully-algorithmic processing with good complexity, while retaining adequate expressivity in modelling common tree updates in HTML5 applications (insofar as detecting redundant CSS rules is concerned). We will start by giving an informal description of our approach and then proceed to our formal model.

3.1 An informal description of the approach

Data representation The data model of HTML5 applications is the standard HTML (DOM) tree. In designing our tree rewriting model, we will adopt unordered (i.e. without a sibling ordering) unranked trees with the set $2^K$ of node labels (where $K$ is a finite set of classes) as a data representation. Not only do unordered trees give a clean data model, but they turn out to be sufficient for analysing redundancy of CSS rules in most HTML5 applications. The choice of tree labels is motivated by CSS and HTML5. Nodes in an HTML document are tagged by HTML elements (e.g. div or a) and associated with a set of classes, which can be added/removed by HTML5 scripts. Node IDs and data attributes are also often assigned to specific nodes, but they tend to remain unmodified throughout the execution of the application and so can conveniently be treated as classes.

An “event-driven” abstraction Our tree-rewriting model is an “event-driven” abstraction of the script component of HTML5 applications. The abstraction consists of a (finite) set of tree-rewrite rules that can be fired any time in any order (so long as they are enabled). In this abstraction, one can imagine that each rewrite rule is associated with an external event listener (e.g. listening for a mouse click, hover, etc.). Since these external events cannot be controlled by the system, it is standard to treat them (e.g. see [33]) as nondeterministic components, i.e., that they can occur at any time in any order. Incidentally, the case for event-driven abstractions has been made in the context of transformations of XML data [14].

A tree-rewrite rule $\sigma$ in our rewrite systems is a tuple $(g, \chi)$ consisting of a node selector $g$ (a.k.a. guard) and a rewrite operation $\chi$. To get a feel for our approach, we will construct an event-driven abstraction for the script component of the HTML5 example in Figure 1. For simplicity, we will now use jQuery calls as tree-rewrite rules. We will formalise them later.

The event-driven abstraction for the example in Figure 1 contains four rewrite rules as follows:

1. $\$(\text{'#limit'})$.addClass('warn');
2. $\$(\text{'.warn'}).removeClass('warn');
3. $\$(\text{'.input_wrap'}).append('<div><input/><a class="remove"></a></div>');
4. $\$(\text{'.input_wrap'}).find('.remove').parent('div').remove();

Note that we removed irrelevant attributes (e.g. href) and text contents since they do not affect our analysis of redundant CSS rules. Rules (1)–(3) were extracted directly from the script. However, the extraction of Rule (4) is more involved. First, the calls to parent() and remove() come directly from the script. Second, the other calls — which select all elements with class remove that are descendants of a node with class input_wrap — derive from the semantics of on(). The connection of the two parts arrives because the jQuery selection is passed to the event handler via the this variable. This connection may be inferred easily by a dataflow analysis that is sensitive to the behaviour of jQuery.

Detecting CSS Redundancy It can be shown that the set $S_1$ of all reachable HTML trees in the example in Figure 1 is a subset of the set $S_2$ of all HTML trees that can be reached by applying Rules (1)–(4) to the initial HTML document. We may detect whether

```
.warn { color: red }
```

is redundant by checking whether its selector may match some part of a tree in $S_2$. If not, then since $S_1 \subseteq S_2$ we can conclude that the rule is definitely redundant. In contrast, if the rule can be matched in $S_2$, we cannot conclude that the rule is redundant in the original application.

Let us test our abstraction. First, by applying Rule (1) to the initial HTML tree, we confirm that warn can appear in a tree in $S_2$ and hence the CSS rule may be fired. We now revisit the scenario in Section 2 where the limit on the number of boxes is removed, but the CSS is not updated. In this case, the new event-driven abstraction for the modified script will not contain Rule (1) and the CSS rule can be seen to be redundant in $S_2$. This necessarily implies that the rule is definitely redundant in $S_1$.

Thus, we guarantee that redundancies will not be falsely identified, but may fail to identify some redundancies in the original application.
### 3.2 Notations for trees

Before defining our formal model, we briefly fix our notations for describing trees. In this paper we use unordered unranked trees. A *tree structure* is a tuple \((D, \leq)\) consisting of a finite subset \(D\) of \(\mathbb{N}^*\) called *tree domain* (where \(\mathbb{N} = \{0, 1, \ldots\}\) and a partial order \(\leq\) over \(D\) satisfying: (1) \(\leq\) is the prefix-of relation, i.e., \(v \leq w\) iff \(w = v\cdot u\) for some \(u \in \mathbb{N}^*\), (2) \(D\) is prefix-closed, i.e., \(w\cdot i \in D\) with \(i \in \mathbb{N}\) implies \(w \in D\).

An *(labeled) tree* over the nonempty finite set (a.k.a. alphabet) \(\Sigma\) is a logical structure of the form \(T = (D, \leq, \lambda)\) where \((D, \leq)\) is a tree structure and \(\lambda\) is a mapping (a.k.a. *node labeling*) from \(D\) to \(\Sigma\). In the sequel, when describing \(T\) we shall omit mention of the relation \(\leq\) and simply write \(T = (D, \lambda)\). We use standard terminologies for trees, e.g., parents, children, ancestors, descendants, and siblings. The *level of a node* \(v \in D\) in \(T\) is \(|v|\). Likewise, the *height of the tree* \(T\) is \(\max\{|v| : v \in D\}\). Let \(\text{Tree}(\Sigma)\) denote the set of trees over \(\Sigma\). For every \(k \in \mathbb{Z}_{\geq 0}\), we define \(\text{Tree}_k(\Sigma)\) to be the set of trees of height \(k\).

If \(T = (D, \lambda)\) and \(v \in D\), the *subtree of \(T\) rooted at \(v\) is the tree* \(T_v = (D', \lambda')\), where \(D' := \{w \in \mathbb{N}^* : vw \in D\}\) and \(\lambda'(w) := \lambda(vw)\).

**Term representations of trees.** As usual, we employ term representations of trees, e.g., \(f(a, g(c, d, e))\) means the tree with root labeled \(f\) with two children labeled \(a\) and \(g\) where the second child in turn has three children labeled \(c, d, e\). In the case when a node has only one child, we remove the bracket for readability, e.g., \(f(a(b))\) can be rewritten as \(f\ a\ b\). In this way, every word \(w \in \Sigma\) has a natural tree representation.

### 3.3 The formal model

We now formally define our tree-rewriting model \(\text{TRS}\) for HTML5 tree updates. A *rewrite system* \(\mathcal{R}\) in \(\text{TRS}\) is a (finite) set of *rewrite rules*. Each rule \(\sigma\) is a tuple \((g, \chi, g', \gamma)\) of a guard \(g\) and a (rewrite) operation \(\chi\). Let us define the notion of guards and rewrite operations in turn.

Our language for guards is simply modal logic with special types of modalities. It is a subset of *tree temporal logic*, which is a formal model of the query language XPath for XML data \([27, 34, 39]\). More formally, a *guard* over the node labeling \(\Sigma = 2^K\) with \(K = \{e_1, \ldots, e_n\}\) can be defined by the following grammar:

\[
g ::= T \mid c \mid g \land g \mid g \lor g \mid \neg g \mid (d)g
\]

where \(c\) ranges over \(K\) and \(d\) ranges over \(\{\uparrow, \uparrow^+, \downarrow, \downarrow^+\}\). The guard \(g\) is said to be *positive* if there is no occurrence of \(\neg\) in \(g\). Given a tree \(T = (D, \lambda)\) and a node \(v \in D\), we define whether \(v\) matches a guard \(g\) (written \(v, T \models g\)) in the standard way, where we interpret \(v, T \models c\) if \((\text{a class } c \in K)\) as \(c \in \lambda(v)\), and each modality \((d)\) (where \(d \in \{\uparrow, \uparrow^+, \downarrow, \downarrow^+\}\)) in accordance with the arrow orientation (possibly with a transitive closure\(^3\)). See the full version for a more formal definition of the semantics and the section below on encoding jQuery rules for some examples. We say that \(g\) is *matched* in \(T\) if \(v, T \models g\) for some node \(v\) in \(T\). Likewise, we say that \(g\) is *matched* in a set \(S\) of \(\Sigma\)-trees if it is matched in some \(T \in S\). In the sequel, we sometimes omit mention of the tree \(T\) from \(v, T \models g\) whenever there is no possibility of confusion.

Having defined the notion of guards, we now define our rewrite operations, which can be one of the following: (1) \(\text{AddChild}(X)\), (2) \(\text{AddClass}(X)\), (3) \(\text{removeClass}(X)\), and (4) \(\text{RemoveNode}\), where \(X \subseteq K\). Intuitively, the semantics of Operations (2)–(4) coincides with the semantics of the jQuery actions \(\text{addClass}(\_\,)\), \(\text{removeClass}(\_\,)\), and \(\text{remove}(\_\,)\), respectively, while the semantics of \(\text{AddChild}(X)\) coincides with the semantics of the jQuery action \(\text{append}([\text{string}])\) in the case when \(\text{string}\) represents a single node associated with classes \(X\). By adding extra classes, appending a larger subtree can be easily simulated by several steps of \(\text{AddChild}(X)\) operations. This is demonstrated in the next section.

We now formally define the semantics of these rewrite operations. Given two trees \(T = (D, \lambda)\) and \(T' = (D', \lambda')\), we say that \(T\) *rewrites* to \(T'\) via \(\sigma\) (written \(T \rightarrow_\sigma T'\)) if there exists a node \(v \in D\) such that \(v \models g\) and

\[
\begin{align*}
\text{if } \chi = \text{AddClass}(X) \text{ then } D' &= D \text{ and } \lambda' := \lambda[v \mapsto X \cup \lambda(v)]\quad \text{ and } \\
\text{if } \chi = \text{AddChild}(X) \text{ then } D' &= D \cup \{v, i\} \text{ and } \lambda' := \lambda[v.i \mapsto X] \text{ where } i \text{ is the number of children of } v \text{ in } T \\
\text{if } \chi = \text{removeClass}(X) \text{ then } D' &= D \text{ and } \lambda' := \lambda[v \setminus \lambda(v)] \quad \text{ and } \\
\text{if } \chi = \text{RemoveNode} \text{ and } v \text{ is not the root node, } D' &= D \setminus \{v, w : w \in \mathbb{N}^*\} \text{ and } \lambda' = \text{the restriction of } \lambda \text{ to } D'.
\end{align*}
\]

Given a rewrite system \(\mathcal{R}\) over \(\Sigma\)-trees, we define \(\rightarrow_\mathcal{R}\) to be the union of \(\rightarrow_\sigma\), for all \(\sigma \in \mathcal{R}\). For every \(k \in \mathbb{Z}_{\geq 0}\), we define \(\rightarrow_{\mathcal{R}, k}\) to be the restriction of \(\rightarrow_\mathcal{R}\) to \(\text{Tree}_k(\Sigma)\). Given a set \(\mathcal{C}\) of \(\Sigma\)-trees, we write \(\text{post}_{\mathcal{R}}(\mathcal{C})\) (resp. \(\text{post}_{\mathcal{R}, k}(\mathcal{C})\)) to be the set of trees \(T'\) satisfying \(T \rightarrow_{\mathcal{R}, k} T'\) (resp. \(T \rightarrow_{\mathcal{R}} T'\)) for some tree \(T \in \mathcal{C}\).

**Encoding jQuery Rewrite Rules** Let us translate the four "jQuery rewrite rules" for the application in Figure 1 into our formalism. The first rule translates directly to the rule \((\#\text{limit}, \text{AddClass}(\{\text{warn}\}))\), while the second rule translates to \((\text{.warn, .removeClass}(\{\text{warn}\}))\). The fourth rule identifies \(\text{div}\) nodes that have some child with class \text{remove} that in turn has some ancestor with class \text{input}_\text{wrap}. Thus, it translates to

\[
(\text{div} \land (\downarrow) . (\text{remove} \land (\uparrow^+). \text{input}_\text{wrap}). \text{RemoveNode}).
\]

---

\(^3\) If desired, we can include extra transitive-reflexive closure modalities \((\downarrow^+)\) and \((\uparrow^+)\) while retaining all upper and lower bound complexity.

\(^4\) Given a map \(f : A \rightarrow B\), \(a' \in A\) and \(b' \in B\), we write \(f[a' \mapsto b']\) to mean the map \(f \setminus \{\langle a', f(a') \rangle\} \cup \{\langle a', b' \rangle\}\).
Finally, the third rule requires the construction of a new subtree. We achieve this through several rules and a new class `tmp`. We first add the new `div` element as a child node, and use the class `tmp` to mark this new node:

\[
\text{(.input\_wrap, AddChild\{div, tmp\})}.
\]

Then, we add the children of the `div` node with the two rules

\[
\text{/(tmp, AddChild\{input\})}.
\]

\[
\text{/(tmp, AddChild\{a, .remove\})}.
\]

**The redundancy problem** The redundancy problem for `TRS` is the problem that, given a rewrite system `R` over `Σ`-trees, a finite nonempty set `S` of guards over `Σ` (a.k.a. guard database), and an initial `Σ`-labeled tree `T_0`, compute the subset `S' ⊆ S` of guards that are not matched in `post_R^*_k(T_0)`.

The decision version of the redundancy problem for `TRS` is simply to check if the aforementioned set `S'` is empty. Similarly, for each `k ∈ Z_{>0}`, we define the `k`-redudancy problem for `TRS` to be the restriction of the redundancy problem for `TRS` to trees of height `k` (i.e. we use `post_R^*_k` instead of `post_R^*_0`).

The problem of identifying redundant CSS node selectors in a CSS file can be reduced to the problem of the redundancy problem for `TRS`. This is because CSS node selectors can easily be translated into our guard language (e.g. using the translation given in [22]). [The converse is false, e.g., CSS selectors cannot select HTML elements `p` with a child. The guard in the third rule of our running example is also not expressible as a CSS selector.] Note that the redundancy problem for `TRS` could have potential applications beyond detecting redundant CSS rules, e.g., detecting redundant jQuery calls in HTML5.

Despite the simplicity of our rewrite rules, it turns out that the redundancy problem is in general undecidable (even restricted to trees of height at most two); see the full version.

**Proposition 1.** The 1-redundancy problem for `R` is undecidable.

4. A monotonic abstraction

The undecidability proof of Proposition 1 in fact relies fundamentally on the power of negation in the guards. A natural question, therefore, is what happens in the case of positive guards. Not only is this an interesting theoretical question, such a restriction often suffices in practice. This is partly because the use of negations in CSS and jQuery selectors (i.e. `:not( . . )`) is rather limited in practice. In particular, there was no use of negations in CSS selectors found in the benchmark in [36] containing 15 live web applications. In practice, negations are almost always limited to negating atomic formulas, i.e., `-c` (for a class `c ∈ K`) which can be overapproximated by `⊤` often without losing too much precision³. A main result of the paper is that the “monotonic” abstraction that is obtained by restricting to positive guards gives us decidability with a good complexity. In this section, we prove the resulting tree rewriting class is “monotonic” in a technical sense of the word, and summarise the main technical results of the paper.

**Notation.** Let us denote by `TRS_0` the set of rewrite systems with positive guards. The guard databases in the input to redundancy and k-redundancy problems for `TRS_0` will only contain positive guards as well. In the sequel, unless otherwise stated, a “guard” is understood to mean a positive guard.

4.1 Formalising and proving “monotonicity”

Recall that a binary relation `R ⊆ S × S` is a preorder if it is transitive, i.e., if `(x, y) ∈ R` and `(y, z) ∈ R`, then `(x, z) ∈ R`. We start with a definition of a preorder `≤` over `TREE(Σ)`, where `Σ = 2^K`. Given two `Σ`-trees `T = (D, λ)` and `T' = (D', λ')`, we write `T ≤ T'` if there exists an embedding from `T'` to `T`, i.e., a function `f : D → D'` such that:

\[
\text{(H1)} f(ε) = ε
\]

\[
\text{(H2) For each } v ∈ D, \lambda(v) ≤ λ'(f(v))
\]

\[
\text{(H3) For each } v.a ∈ D \text{ where } v ∈ N^* \text{ and } a ∈ N, \text{ we have } f(v.a) = f(v).b \text{ for some } b ∈ N.
\]

Note that this is equivalent to the standard notion of homomorphisms from database theory (e.g. see [11]) when each class `c ∈ K` is treated as a unary relation. The following is a basic property of `≤`, whose proof is easy and is left to the reader.

**Fact.** `≤` is a preorder on `TREE(Σ)`.

As an aside, the non-reflexive version `≺` of `≤` is a well-
quasi-order since it does not allow an infinite descending chain `T_1 ≻ T_2 ≻ ⋅ ⋅ ⋅`. The following lemma shows that embeddings preserve positive guards.

**Lemma 2.** Given trees `T = (D, λ)` and `T' = (D', λ') ∈ TREE(Σ)` satisfying `T ≤ T'` with a witnessing embedding `f : D → D'`, and a positive guard `g` over `Σ`, then if `v, T ⊨ g` then `f(v), T' ⊨ g`.

We relegate to the full version the proof of Lemma 2, which is similar to (part of) the proof of the homomorphism theorem for conjunctive queries (e.g. see [11, Theorem 6.2.3]). This lemma yields the following monotonicity property of `TRS_0`.

**Lemma 3 (Monotonicity).** For each `σ ∈ R`, if `T_1 ≧ T_2` and `T_1_σ T'_1 ≧ T_2` then either `T'_1 ≧ T_2` or `T_2_σ T'_2 for some `T'_2` satisfying `T'_1 ≧ T'_2`.

³ In the case when `c` is an HTML tag name (e.g. `div` or `img`), then we can overapproximate this with the positive formula `\/c ∈ C\{c\}(ε)`, where `C` is the set of all HTML tag names in the input HTML5 application, without losing any precision.
Intuitively, the property states that any rewriting step of the “smaller” tree can be simulated by the “bigger” tree while still preserving the embedding relation. The proof of the lemma is easy (by considering all four possible rewrite operations), and is relegated to the full version.

One consequence of this monotonicity property is that, when dealing with the redundancy problem, we can safely ignore rewrite rules that use the rewrite operation RemoveNode or RemoveClass(X). This is formalized in the following lemma, whose proof is given in the full version.

Lemma 4. Given a rewrite system $\mathcal{R}$ over $\Sigma$-trees, a guard database $S$, and an initial tree $T_0$, let $\mathcal{R}^-$ be the set of $\mathcal{R}$-rules less that use RemoveNode and RemoveClass(X). Then, for each $g \in S$, $g$ is matched in $post^*_\mathcal{R}(T_0)$ iff $g$ is matched in $post^*_{\mathcal{R}^-}(T_0)$.

Convention. In the sequel, we assume that there are only two possible rewrite operations, namely, AddChild(X), and AddClass(X).

4.2 Summary of technical results

We have completely identified the computational complexity of the redundancy and $k$-redundancy problem for TRS$_0$. Our first result is:

Theorem 1. The redundancy problem for TRS$_0$ is EXP-complete.

Although obtaining decidability per se is not difficult (e.g. by using Lemma 3 and the theory of well-structured transition systems [9, 18], which only gives a nonelementary complexity), obtaining a tight complexity bound for the redundancy problem for TRS$_0$ is non-trivial, which is our main contribution. Moreover, our upper bound was obtained via an efficient reduction to an analysis of symbolic pushdown systems (see Section 5), for which there are highly optimised tools (e.g. Bobop [15], Getafix [25], and Moped [38]). We have implemented our reduction and demonstrate its viability in detecting redundant CSS rules in HTML5 applications (see Section 6). The proof of the lower bound in Theorem 1 is provided in the full version. In the case of $k$-redundancy problem, a better complexity can be obtained.

Theorem 2. The $k$-redundancy problem TRS$_0$ is:

(i) PSPACE-complete if $k$ is part of the input in unary.
(ii) solvable in P-time — $n^{O(k)}$ — for each fixed parameter $k$, but is $\mathcal{W}[1]$-hard.

The $\mathcal{W}[1]$-hardness in this theorem is evidence from parameterised complexity theory [19] which suggests that our $n^{O(k)}$-time algorithm is essentially optimal for each fixed height $k$. For space reasons, we relegate the proofs of Theorem 2 to the full version.

5. The algorithm

In this section, we provide an efficient reduction to an analysis of symbolic pushdown systems, which will give an exponential-time (resp. polynomial-space) algorithm for the redundancy (resp. $k$-redundancy) problem for TRS$_0$. To this end, we first provide a preliminary background on symbolic pushdown systems. We will then provide a roadmap of our reduction to symbolic pushdown systems, which will consist of a sequence of three polynomial-time reductions described in the last three subsections.

5.1 Pushdown systems: a preliminary

Before describing our reduction, we will first provide a preliminary background on pushdown systems and their extensions to symbolic pushdown systems.

Pushdown systems are standard pushdown automata without input labels. Input labels are irrelevant since one mostly asks about their transition graphs (in our case, reachability). More formally, a pushdown system (PDS) is a tuple $P = (\mathcal{Q}, \Gamma, \Delta)$, where $\mathcal{Q}$ is a finite set of control states, $\Gamma$ is a finite set of stack symbols, and $\Delta$ is a finite subset of $(\mathcal{Q} \times \Gamma) \times (\mathcal{Q} \times \Gamma^*)$ such that if $((q, a), (q', w)) \in \Delta$ then $|w| \leq 2$. This PDS generates a transition relation $\rightarrow_P \subseteq (\mathcal{Q} \times \Gamma^*) \times (\mathcal{Q} \times \Gamma^*)$ as follows: $(q, v) \rightarrow_P (q', v')$ if there exists a rule $((q, a), (q', w)) \in \Delta$ such that $v = u a$ and $v' = u w$ for some word $u \in \Gamma^*$.

Symbolic pushdown systems are pushdown systems that are succinctly represented by boolean formulas. More precisely, a symbolic pushdown system (sPDS) is a tuple $(\mathcal{V}, \mathcal{W}, \Delta)$, where $\mathcal{V} = \{x_1, \ldots, x_n\}$ and $\mathcal{W} = \{y_1, \ldots, y_m\}$ are two disjoint sets of boolean variables, and $\Delta$ is a finite set of pairs $(i, \phi)$ of number $i \in \{0, 1, 2\}$ and boolean formula $\phi$ over the set of variables $\mathcal{V} \cup \mathcal{W} \cup \mathcal{V}^* \cup \mathcal{W}^*$, where $\mathcal{V}^* := \{x_1^*, \ldots, x_n^*\}$ and $\mathcal{W}^* := \bigcup_{j=1}^m W_j$ where $W_j := \{y_1^j, \ldots, y_m^j\}$. This sPDS generates a (exponentially bigger) pushdown system $P = (\mathcal{Q}, \Gamma, \Delta')$, where $\mathcal{Q} = \{0, 1\}^n$, $\Gamma = \{0, 1\}^m$, and $((q, a), (q', w)) \in \Delta'$ iff there exists a pair $(i, \phi) \in \Delta$ satisfying $i = |w|$, and $\phi$ is satisfied by the assignment that assigns $6$ to $\mathcal{V}$, $a$ to $\mathcal{W}$, $q'$ to $\mathcal{V}^*$, and $w$ to $\mathcal{W}^*$ (i.e. assigning the $j$th letter of $w$ to $W_j$). As we will show in the full version, using a BDD (Binary Decision Diagram) representation of boolean formulas is adequate for our purposes. BDD representations are crucial since they will allow us to tap into efficient sPDS solvers which exploit BDD representations (e.g. Moped [38, 40]).

The bit-toggling problem for sPDS is a simple reachability problem over symbolic pushdown systems: given an sPDS $P = (\mathcal{V}, \mathcal{W}, \Delta)$ with $\mathcal{V} = \{x_1, \ldots, x_n\}$ and $\mathcal{W} = \{y_1, \ldots, y_m\}$, a variable $y_i \in \mathcal{W}$, and an initial configuration $I_0 = ((b_1, \ldots, b_n), (b'_1, \ldots, b'_m)) \in \{0, 1\}^n \times \{0, 1\}^m$, decide if $I_0 \rightarrow_P^* (q, a)$ for some $q \in \{0, 1\}^n$ and $a = (b''_1, \ldots, b''_m) \in \{0, 1\}^m$ with $b''_i = 1$. In other words, we

6 Meaning that if $q = (q_1, \ldots, q_n)$, then $q_i$ is assigned to $x_i$
want to decide if we can toggle on the variable $y_i$ from the initial configuration. The bounded bit-toggling problem is the same as the bit-toggling problem but the stack height of the pushdown system cannot exceed some given input parameter $h \in \mathbb{N}$ (given in unary).

**Proposition 5.** The bit-toggling (resp. bounded bit-toggling) problem for $sPDS$ is solvable in EXP (resp. PSPACE).

The proof of this is standard (e.g. see [40]), which for completeness we provide in the full version.

### 5.2 Intuition/Roadmap of the reduction

The following theorem formalises our reduction claim.

**Theorem 3.** The redundancy (resp. $k$-redundancy) problem for $TRS_0$ is polynomial-time reducible to the bit-toggling (resp. bounded bit-toggling) problem for $sPDS$.

Together with Proposition 5, Theorem 3 implies an EXP (resp. PSPACE) upper bound for the redundancy (resp. $k$-redundancy) problem for $sPDS$. Moreover, as discussed in the full version, it is straightforward to construct from our reduction a counterexample path in the rewrite system witnessing the non-redundancy of a given guard.

We now provide a high-level proof idea of Theorem 3. Given a rewrite system $R \in TRS_0$, an initial tree $T = (D, \lambda) \in TREE(\Sigma)$ with $\Sigma = 2^K$, and a guard database $S$, we try to “saturate” each node $v \in D$ with the classes that may be added to $v$. i.e., if $T_1 = (D_1, \lambda_1)$ (resp. $T_2 = (D_2, \lambda_2)$) is the tree before (resp. after) applying a saturation step, then $D_1 = D_2$ (i.e. no nodes are added) and there exists a node $v \in D$ such that $\lambda_1(v) \subset \lambda_2(v)$ (i.e. some classes are added to $\lambda_1(v)$). In particular, we have $T_1 \prec T_2$. There are two saturation rules that are repeatedly applied until we reach a fixpoint. The first saturation rule is to apply a rewrite rule $(g, AddClass(X)) \in R$ at $v$. The second saturation rule is more involved. We construct a pushdown system $P$ starting with an initial stack of height 1 with content $\lambda(v)$ (possibly with a some extra “context” information) and “simulate” each possible branch that is spawned from $v$. The pushdown system is an $sPDS$ that keeps one boolean variable and only check redundancy at the root node. We then show that the simplified problem is polynomial-time solvable assuming oracle calls to the “class-adding (reachability) problem”, a simple reachability problem involving only single-node input trees possibly with a parent node that only provides a “static context” (i.e. cannot be modified). Finally, we show that the class-adding problem is efficiently reducible to the bit-toggling problem for $sPDS$. The case of $k$-redundancy for $TRS_0$ is similar but each intermediate problem is relatived to the version with bounded height.

### 5.3 Simplifying the rewrite system

We will make two simplifications: (1) restricting the problem to only checking redundancy at the root node, (2) restrict the guards to be used.

To achieve simplification (1), one can simply define a new set $S'$ of guards from $S$ as follows: $S' = \{g \vee (\downarrow^+ g) : g \in S\}$. Then, for each tree $T = (D, \lambda) \in TREE(\Sigma)$ and guard $g$, it is the case that $(\exists v \in D : v, T \models g) \iff (\exists v \in D : v, T \models (\downarrow^+ g))$. We now proceed to simplification (2). A guard over the node labeling $\Sigma = 2^K$ is said to be simple if it is of the form $\bigwedge_{i=1}^{m} c_i$ or $\langle d \rangle \bigwedge_{i=1}^{m} c_i$ for some $m \in \mathbb{N}$, where each $c_i$ ranges over $K$ and $d$ ranges over $\{\top, \bot\}$. [Note: if $m = 0$, then $\bigwedge_{i=1}^{m} c_i = \top$.] For notational convenience, if $X = \{c_1, \ldots, c_m\}$, we shall write $X$ (resp. $\langle d \rangle X$) to mean $\bigwedge_{i=1}^{m} c_i$ (resp. $\langle d \rangle \bigwedge_{i=1}^{m} c_i$). A rewrite system $R \in TRS_0$ is said to be simple if (i) all guards occurring in $R$ are simple, and (ii) if $(\langle d \rangle X, \chi) \in R$, then $\chi$ is of the form $AddClass(Y)$. We define $TRS_0 \subseteq TRS_0$ to be the set of simple rewrite systems. The redundancy (resp. $k$-redundancy) problem for $TRS_0$ is defined in the same way as for $TRS_0$ except that all the guards in the guard database are restricted to be a subset of $K$. The following lemma shows that the redundancy (resp. $k$-redundancy) problem for $TRS_0$ can be reduced in polynomial time to the redundancy (resp. $k$-redundancy) problem for $TRS_0'$.

**Lemma 6.** Given a $R \in TRS_0$ over $\Sigma = 2^K$ and a guard database $S$ over $\Sigma$, there exists $R' \in TRS_0'$ over $\Sigma' = 2^{K'}$ (where $K \subseteq K'$) and a set $S' \subseteq K'$ of simple guards such that:

1. $R$ is $k$-redundant for $R'$ iff $R'$ is $k$-redundant for $R$.
2. $S$ is redundant for $R$ iff $S'$ is redundant for $R'$.

Moreover, we can compute $R'$ and $S'$ in polynomial time.

We show how to compute $R'$. The set $K'$ is defined as the union of $K$ with the set $G$ of all subformulas (i.e. occurring in the parse tree) of guard formulas in $S$ and $R$. In the sequel, to avoid potential confusion, we will often underline members of $G$ in $K'$, e.g., write $\downarrow c$ instead of $\downarrow^+ c$. We now define the simple rewrite system $R'$. Initially, we will define a rewrite system $R_1$ that allows the operators $\langle \downarrow^+ \rangle$ and $\langle \downarrow \rangle$; later we will show how to remove them. We first add the following “intermediate” rules to $R_1$:

1. $(\{g, g'\}, AddClass(g \land g'))$, for each $(g \land g') \in G$. 


2. \((g, \text{AddClass}(g \lor g'))\) and \((g', \text{AddClass}(g \lor g'))\), for each \((g \lor g') \in G\).
3. \(((d)g, \text{AddClass}((d)g))\), for each \((d)g \in G\).
4. \((g, \chi)\), for each \((g, \chi) \in R\).

Note that in Rule (3) the guard \((d)g\) is understood to mean a non-atomic guard over \(2^K\). Finally, we define \(S' := \{g : g \in S\}\). Notice that each guard in \(S'\) is atomic. The aforementioned algorithm computes \(R_1\) and \(S'\) in linear time.

We now show how to remove the operators \(\langle \uparrow + \rangle\) and \(\langle \downarrow + \rangle\) (i.e. rules of type (3)). The resulting rewrite system will be our final rewrite system \(R'\). Initially, we set \(R' := R_1\). Next, for each rule \(((d^+)g, \chi)\) in \(R_1\) where \(d \in \{\uparrow, \downarrow\}\), we add the following rules to \(R'\):

(a) \(((d)g, \text{AddClass}((d^+)g))\).
(b) \(((d)(d^+)g, \text{AddClass}((d^+)g))\).

Note that \(R_1\) contains the rule \(((d^+)g, \chi)\), where \((d^+)g\) is understood to mean an atomic guard over \(2^K\). Intuitively, this simplification can be done because \(v, T \models (d^+)g\) iff at least one of the following cases holds: (i) \(v, T \models (d)g\), (ii) there exists a node \(w\) in \(T\) such that \(w, T \models (d^+)g\) and \(w\) can be reached from \(v\) by following the direction \(d\) for one step. The aforementioned rewrite step again can be done in linear time. The proof of correctness (i.e. \((P1)\) and \((P2)\)) is provided in the full version. In particular, for all \(g \in S\), it is the case that \(g\) is redundant in \(R\) iff \(g\) is redundant in \(R'\).

### 5.4 Redundancy → class-adding

We will show that redundancy for \(\text{TRs}_0\) can be solved in polynomial time assuming an oracle to the “class adding problem” for \(\text{TRs}_0\). The class adding problem is a reachability problem for \(\text{TRs}_0\) involving only single-node input trees possibly with a parent node that only provides a “context” (i.e. cannot be modified). As we will see in the following subsection, the class-adding problem for \(\text{TRs}_0\) lends itself to a fast reduction to the bit-toggling problem for \(s\text{PDS}\).

Similarly, \(k\)-redundancy can be solved via the same routine, where intermediate problems are restricted to their bounded height equivalents.

Before formally defining the class-adding problem, we first need the definition of an “assumption function”, which plays the role of the possible parent context node but can be treated as a separate entity from the input tree. As we shall see, this leads to a more natural formulation of the computational problem. More precisely, an assumption function \(f\) over the alphabet \(\Sigma = 2^K\) is a function mapping each element of \(\{\text{root}\} \cup K\) to \(\{0, 1\}\). The boolean value of \(f(\text{root})\) is used to indicate whether the input single-node tree is a root node. Given a tree \(T = (D, \lambda) \in \text{Tree}(\Sigma)\), a node \(v \in D\), and a simple guard \(g\) over \(\Sigma\), we write \(v, T \models_f g\) if one of the following three cases holds: (i) \(v \neq \epsilon\) and \(v, T \models g\), (ii) \(v = \epsilon\), \(g\) is not of the form \(\langle \uparrow \rangle X\), and \(v, T \models g\), and (iii) \(v = \epsilon\), \(g = \langle \uparrow \rangle X\), \(f(\text{root}) = 0\), and \(X \subseteq \{c \in K : f(c) = 1\}\). In other words, \(v, T \models_f g\) checks whether \(g\) is satisfied at node \(v\) assuming the assumption function \(f\) (in particular, if \(f(\text{root}) = 0\), then any guard referring to the parent of the root node of \(T\) is checked against \(f\)).

Given a simple rewrite system \(R \in \text{TRs}_0\) over \(\Sigma\) and an assumption function \(f\), we may define the rewriting relation \(\rightarrow_R \subseteq \text{Tree}(\Sigma) \times \text{Tree}(\Sigma)\) in the same way as we define \(\rightarrow\), except that \(\models_f\) is used to check guard satisfaction. The class-adding (reachability) problem is defined as follows: given a single-node tree \(T_0 = (\{\epsilon\}, \lambda_0) \in \text{Tree}(\Sigma\) with \(\Sigma = 2^K\), an assumption function \(f : (\{\text{root}\} \cup K) \rightarrow \{0, 1\}\), a class \(c \in K\), and a simple rewrite system \(R\), decide if there exists a tree \(T = (D, \lambda)\) such that \(T_0 \rightarrow^*_R T\) and \(c \in \lambda(\epsilon)\). Similarly, the \(k\)-class-adding problem is defined in the same way as the class-adding problem except that the reachable trees are restricted to height \(k\) (\(k\) is part of the input).

**Lemma 7.** The redundancy (resp. \(k\)-redundancy) problem for simple rewrite systems is \(P\)-time solvable assuming oracle calls to the class-adding (resp. \(k\)-class-adding) problem.

Given a tree is \(T_0 = (D_0, \lambda_0) \in \text{Tree}(\Sigma)\) with \(\Sigma = 2^K\), a simple rewrite system \(R\) over \(\Sigma\), and a set \(S \subseteq K\), the task is to decide whether \(S\) is redundant (or \(k\)-redundant). We shall give the algorithm for the redundancy problem; the \(k\)-redundancy problem can be obtained by simply replacing oracle calls to the class-adding problem by the \(k\)-class-adding problem.

The algorithm is a fixpoint computation. Let \(T = (D, \lambda) := T_0\). At each step, we can apply any of the following “saturation rules”:

- If \((g, \text{AddClass}(B))\) is applicable at a node \(v \in D_0\) in \(T\), then \(\lambda(v) := \lambda(v) \cup B\).
- If the class-adding problem has a positive answer on input \((T_v, f, c, R)\), then \(\lambda(v) := \lambda(v) \cup \{c\}\), for some \(v \in D, c \in K \setminus \lambda(v)\), and \(T_v := (\epsilon, \lambda_v)\) with \(\lambda_v(c) = \lambda(v)\), where we define \(f : (\{\{\} \cup K) \rightarrow \{0, 1\}\) with \(f(\text{root}) = 0 \iff v = \epsilon\) and, if \(f(\text{root}) = 0\) and \(u\) is the parent of \(v\), then \(f(u) = 1 \iff a \in \lambda(\epsilon)\).

Observe that saturation rules can be applied at most \(|K| \times |D|\) times. Therefore, when they can be applied no further, we check whether \(S \cap \lambda(\epsilon) \neq \emptyset\) and terminate. Assuming constant-time oracle calls to the class-adding problem, the algorithm easily runs in polynomial time. Furthermore, since each saturation rule only adds new classes to a node label, the correctness of the algorithm can be easily proven using Lemma 2 and Lemma 3; see the full version.

### 5.5 Class-adding → bit-toggling

**Lemma 8.** The class-adding (resp. \(k\)-class-adding) problem for \(\text{TRs}_0\) is polynomial-time reducible to the bit-toggling (resp. bounded bit-toggling) problem for \(s\text{PDS}\).
We prove the lemma above. Fix a simple rewrite system \( \mathcal{R} \) over the node labeling \( \Sigma = 2^K \), an assumption function \( f : \{ \text{root} \} \cup \mathcal{K} \to \{ 0, 1 \} \), a single node tree \( T_0 = (\{ \epsilon \}, \lambda_0) \in \text{Tree}(\Sigma) \), and a class \( \alpha \in \mathcal{K} \).

We construct an sPDS \( \mathcal{P} = (\mathcal{V}, \mathcal{W}, \Delta) \). Intuitively, the sPDS \( \mathcal{P} \) will simulate \( \mathcal{R} \) by exploring all branches in all trees reachable from \( T_0 \) while accumulating the classes that are satisfied at the root node. Define \( \mathcal{V} := \{ x_c : c \in \mathcal{K} \} \cup \{ \text{pop} \} \), and \( \mathcal{W} := \{ y_{c,z} : c \in \mathcal{K} \} \cup \{ \text{root} \} \). Roughly speaking, we will use the variable \( y_{c,z} \) (resp. \( z_{c} \)) to remember whether the class \( c \) is satisfied at the current (resp. parent of the current) node being explored. The variable \( x_{c} \) is needed to remember whether the class \( c \) is satisfied at a child of the current node (i.e., after a pop operation). The variable \( \text{root} \) signifies whether the current node is a root node, while the variable \( \text{pop} \) indicates whether the last operation that changed the stack height is a pop.

We next define \( \Delta \):

- For each \((A, \text{AddClass}(B)) \in \mathcal{R}\), let \( C := \mathcal{K} \setminus (A \cup B) \). Now add the rule \((1, \varphi)\), where \( \varphi \) is a conjunction of: (a) \( \text{pop} \leftrightarrow \text{pop}' \), (b) \( \text{root} \leftrightarrow \text{root}' \), (c) \( \bigwedge_{c \in C}(y_c \leftrightarrow y'_c) \), (d) \( \bigwedge_{a \in A}(y_a \land y_{a,0}) \), (e) \( \bigwedge_{b \in B}(y_b) \), (f) \( \bigwedge_{c \in C}(z_c \leftrightarrow z'_c) \), and (g) \( \bigwedge_{c \in K}(y_c \leftrightarrow y'_c) \).
- For each \((A, \text{AddChild}(B)) \in \mathcal{R}\), add the rule \((2, \varphi)\), where \( \varphi \) is a conjunction of: (a) \( \neg \text{pop}' \land \neg \text{root}' \land \text{root} \leftrightarrow \text{root}' \), (b) \( \bigwedge_{c \in K}(y_c \leftrightarrow y'_c) \), (c) \( \bigwedge_{a \in A}(y_a \land y_{a,0} \land z_{a,1}) \), (d) \( \bigwedge_{b \in B}(y_b) \), and (e) \( \bigwedge_{c \in K} z_{c} \leftrightarrow z'_{c} \).
- For each \((\mathcal{A}, \text{AddClass}(B)) \in \mathcal{R}\), add the rule \((1, \varphi)\), where \( \varphi \) is a conjunction of: (a) \( \text{pop} \leftrightarrow \text{pop}' \), (b) \( \text{root} \leftrightarrow \text{root}' \), (c) \( \bigwedge_{c \in K}(x_c \leftrightarrow x'_c) \), (d) \( \bigwedge_{a \in A}(x_a \land x_{a,0}) \), (e) \( \bigwedge_{c \in K}(z_c \leftrightarrow z'_c) \), (f) \( \bigwedge_{b \in B}(y_b) \), and (g) \( \bigwedge_{c \in K}(y_c \leftrightarrow y'_c) \).
- For each \((\mathcal{A}, \text{AddChild}(B)) \in \mathcal{R}\), add the rule \((2, \varphi)\), where \( \varphi \) is a conjunction of: (a) \( \text{pop} \leftrightarrow \text{pop}' \), (b) \( \text{root} \leftrightarrow \text{root}' \), (c) \( \bigwedge_{a \in A}(x_a \land x_{a,0}) \), (d) \( \bigwedge_{b \in B}(y_b) \), and (e) \( \bigwedge_{c \in K} z_{c} \leftrightarrow z'_{c} \).
- Finally, add the rule \((0, \varphi)\), where \( \varphi \) is a conjunction of: (a) \( \text{pop}' \land \neg \text{root} \), (b) \( \bigwedge_{c \in K}(y_c \leftrightarrow y'_c) \).

These boolean formulas can easily be represented as BDDs (see full version).

Continuing with our translation, the bit that needs to be toggled on is \( y_a \). We now construct the initial configuration for our bit-toggling problem. For each subset \( X \subseteq \mathcal{K} \) and a function \( q : \mathcal{V} \to \{ 0, 1 \} \), define the function \( I_{X,f,q} : (\mathcal{V} \cup \mathcal{W}) \to \{ 0, 1 \} \) as follows: \( I_{X,f,q} (\text{root}) := f(\text{root}) \), \( I_{X,f,q} (\text{pop}) := q(\text{pop}) \), and for each \( c \in \mathcal{K} \): (i) \( I_{X,f,q}(x_c) := q(x_c) \), (ii) \( I_{X,f,q}(y_c) = 1 \) if \( c \in X \), and (iii) \( I_{X,f,q}(z_c) := f(c) \). We shall write \( I_{X,f} \) to mean \( I_{X,f,q} \) with \( q(x) = 0 \) for each \( x \in \mathcal{V} \). Define the initial configuration \( I_0 \) as the function \( I_{\alpha,0,e,f} \).

Let us now analyse our translation. The translation is easily seen to run in polynomial time. In fact, with a more careful analysis, one can show that the output sPDS is of linear size and that the translation can be implemented in polynomial time. Correctness of our translation immediately follows from the following technical lemma:

**Lemma 9.** For each subset \( X \subseteq \mathcal{K} \), the following are equivalent:

(A1) There exists a tree \( T = (D, \lambda) \in \text{Tree}(\Sigma) \) such that \( T_0 \rightarrow^* R T \) and \( \lambda(c) = X \).

(A2) There exists \( q' : \mathcal{V} \to \{ 0, 1 \} \) such that \( I_{\alpha,0,e,f} \rightarrow^{\mathcal{P}}_{q'} I_{X,f,q'} \).

This lemma intuitively states that the constructed sPDS performs a "faithful simulation" of \( \mathcal{R} \). Moreover, \((A2) \Rightarrow (A1)\) gives soundness of our reduction, while \((A1) \Rightarrow (A2)\) gives completeness of our reduction. The proof is very technical, which we relegate to the full version.

### 6. Experiments

We have implemented our approach in a new tool TreePed which is available for download [5]. We tested it on several case studies. Our implementation contains two main components: a proof-of-concept translation from HTML5 applications using jQuery to our model, and a redundancy checker (with non-redundancy witness generation) for our model. Both tools were developed in Java. The redundancy checker uses jMoped [42] to analyse symbolic pushdown systems. In the following sections we discuss the redundancy checker, translation from jQuery, and the results of our case studies.

#### 6.1 The Redundancy Checker

The main component of our tool implements the redundancy checking algorithm for proving Theorem 1. Largely, the algorithm is implemented directly. The most interesting differences are in the use of jMoped to perform the analysis of sPDSs to answer class adding checks. In the following, assume a tree \( T_c \), rewrite rules \( \mathcal{R} \), and a set \( \mathcal{K} \) of classes.

**Optimising the sPDS** For each class \( c \in \mathcal{K} \) we construct an sPDS. We can optimise by restricting the set of rules in \( \mathcal{R} \) used to build the sPDS. In particular, we can safely ignore all rules in \( \mathcal{R} \) that cannot appear in a sequence of rules leading to the addition of \( c \). To do this, we begin with the set of all rules that either directly add the class \( c \) or add a child to the tree (since these may lead to new nodes matching other rules). We then add all rules that directly add a class \( c' \) that appears in the guard of any rules included so far. This is iterated until a fixed point is reached. The rules in the fixed point are the rules used to build the sPDS.

**Reducing the number of calls** We reimplemented Moped’s global backwards reachability analysis (and witness generation) in [40] in jMoped. This means that a single call to jMoped can allow us to obtain a BDD representation
of all initial configurations of the sPDSs obtained from class adding problems \((T_V, f, c, R)\) that have a positive answer to the class-adding problem for a given class \(c \in \mathcal{K}\). Thus, we only call jMoped once per class.

6.2 Translation from HTML5

The second component of our tool provides a proof-of-concept prototypical translation from HTML5 using jQuery to our model. There are three main parts to the translation.

- The DOM tree of the HTML document is directly translated to a tree in our model. We use classes to encode element types (e.g. `div` or `a`), IDs and CSS classes.
- We support a subset of CSS covering the most common selectors and all selectors in our case studies. Each selector in the CSS stylesheet is translated to a guard in the guard database. For pseudo-selectors such as `g:hover` and `g:before` we simply check for the redundancy of `g`.
- Dynamic rules are extracted from the JavaScript in the document by identifying jQuery calls and generating rules as outlined in Section 3.3. Developing a translation tool that covers all covers all aspects of such an extremely rich and complex language as JavaScript is a difficult engineering problem. Our proof-of-concept prototype covers many, but by no means all, interesting features of the language. The implemented translation is described in more detail in the full version.

Sites formed of multiple pages with common CSS files are supported by automatically collating the results of independent page analyses and reporting site-wide redundancies.

6.3 Case Studies

We performed several case studies. The first is based on the Igloo example from the benchmark suite of the dynamic CSS analyser Cilla [36], and is described in detail below. The second (and largest) is based on the Nivo Slider plugin [41] for animating transitions between a series of images. The remaining examples are hand built and use jQuery to make frequent additions to and removals from the DOM tree. The first `bikes.html` allows a user to select different frames, wheels and groupsets to build a custom bike, `comments.html` allows a user to select different frames, wheels and groupsets to build a custom bike, `comments.html` displays a comments section that is loaded dynamically via an AJAX call, and `transactions.html` is a finance page where previous transactions are loaded via AJAX and new transactions may be added and removed via a form. The example in Figure 1 is `example.html` and `example-up.html` is the version without the limit on the number of input boxes. These examples are available in the `src/examples/html` directory of the tool distribution [5].

All case studies contained non-trivial CSS selectors whose redundancy depended on the dynamic behaviour of the system. In each case our tool constructed a rewrite system following the process outlined above, and identified all redundant rules correctly.

<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>23</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>3</td>
<td>1 (1)</td>
<td>15</td>
<td>3</td>
<td>.6s</td>
</tr>
<tr>
<td>igloo/</td>
<td>261 (89)</td>
<td>24</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index.html</td>
<td>145</td>
<td>24</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>engineering.html</td>
<td>236</td>
<td>24</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nivo-Slider/</td>
<td>236</td>
<td>24</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>demo.html</td>
<td>236</td>
<td>24</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Case study results.

The experiments were run on a Dell Latitude e6320 laptop with 4Gb of RAM and four 2.7GHz Intel i7-2620M cores. We used OpenJDK 7, using the argument “-Xmx” to limit RAM usage to 2.5Gb. The results are shown in Table 1. Ns is the initial number of elements in the DOM tree, Ss is the number of CSS selectors (with the number of redundant selectors shown in brackets), Ls is the number of Javascript lines reported by cloc, and Rs is the number of rules in the rewrite system obtained from the JavaScript after simplification (unsimplified rules may have arbitrarily complex guards). The figures for the Igloo example are reported per file or for the full analysis as appropriate.

**The Igloo example** The Igloo example is a mock company website with a home page (`index.html`) and an engineering services page (`engineering.html`). The page includes a search bar that contains some placeholder text which is present only when the search bar is empty and does not have focus. Placeholder text is supported by some browsers, but not all, and the page contains a small amount of JavaScript to this functionality when it is not provided by the browser. In particular, the CSS class `touched`, and rule

```javascript
#search .touched { color: #333; }  
```

are used for this purpose. Since the JavaScript is not executed in most browsers, Cilla incorrectly claims the rule is redundant. Since we only identify genuinely redundant rules, our tool correctly does not report the rule as redundant.

In addition, TreePed identified a further unexpected mistake in Igloo’s CSS. The rule

```javascript
h2 a:hover, h2 a:active, h2 a:focus  
```

is missing a comma from the end of the first line. This results in the redundant selector “h2 a:focus h3 a:hover” rather than two separate selectors as was intended. Cilla did not report this redundancy as it appears to ignores all CSS rules with pseudo-selectors. Finally, we remark that the second line of the above rule contains a further error: “h2 a:focus” should in fact be “h3 a:focus”. This raises the question of selector subsumption, on which there are al-

---

Not including the number of rules required to represent the CSS selectors.
ready a lot of research works (e.g. see [16, 36]). These algorithms may be incorporated into our tool to obtain a further size reduction of CSS files.

7. Future Work

At the moment our translation from HTML5 applications to our tree-rewriting class is prototypical and does incorporate many features that such a rich and complex language as JavaScript has. An important research direction is to extend our translation to allow more features, ultimately allowing the tool to work on most live HTML5 applications. Another direction is to find better ways of overapproximating redundancy problems for the undecidable class TRS of rewrite systems using our monotonic abstractions (other than replacing non-positive guards by $\top$). In particular, for a general guard, can we effectively (and efficiently) construct the “most precise” positive guard that serves as an overapproximation?

Acknowledgments

We sincerely thank Max Schaefer for fruitful discussions. Hague is supported by the Engineering and Physical Sciences Research Council [EP/K009907/1]. Lin is supported by a Yale-NUS Startup Grant. Ong is partially supported by a visiting professorship, National University of Singapore.

References