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

Matthew Hague[1], Anthony Widjaja Lin[2], and C.-H. Luke Ong[3]

[1] Royal Holloway, University of London
[2] Yale-NUS College
[3] 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 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.

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 [35] 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 [36] 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. [34, 36, 35, 21, 15]), which include merging "duplicated" CSS rules, refactoring CSS declaration blocks, and simplifying CSS selectors, to name a few. However, most of these works analyse 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 general test all possible behaviors of the HTML5 application. For example, from the benchmarks of [35] there are some non-redundant CSS rules that their tool CILLA 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. [31, 16, 11, 20, 19, 12]) and infinite-state verification (e.g. [22, 28, 29, 30, 9, 27]) — 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 [40] and real-world examples from the benchmark in [35]), 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* [29] and their extensions [30, 28, 22] (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. [11, 19, 20, 12]) 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. Moreoever, our algorithm relies on an efficient reduction to a reachability analysis in *symbolic pushdown systems* for which highly optimised solvers (e.g. Bebop

[14], Getafix [24], and Moped [37]) are available.

We have implemented our reduction, together with a proof-of-concept[1] translation tool from HTML5 to our tree rewriting model. Our translation specifically focuses on modelling standard features of *jQuery* [23] — 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 [1]) some authors [25] 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 [35].

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 appendix.

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 [2]). We will discuss their formal models in Section 3. Our example (also available at the URL [6]) is a small modification of an example taken from an online tutorial [3], which is given 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 [23] 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

[1] Handling JavaScript in its full generality is a difficult engineering problem, which is beyond the scope of this paper.

```
<html>
<head><style>.warn { color: red }</style></head>
<body>
    <div class="input_wrap">
        <button class="button">Add Fields</button>
        <div><input type="text" name="mytext[]"> </div>
    </div>
    <div>Total: </div><div id="counter">1</div>
    <div id="limit">Limits not reached</div>

    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js">
    </script>
    <script type="text/javascript">
        $(document).ready(function() {
            var x = 1;

            $('.button').click(function(e){
                if(x < 10){
                    x++;
                    $('.input_wrap').append(
                      '<div>' +
                      '    <input type="text" name="mytext[]"/>' +
                      '    <a href="#" class="remove">Remove</a>' +
                      '</div>'
                    );
                    $('#counter').html(x);
                }
                else {
                    $('#limit').html('Limits reached');
                    $('#limit').addClass('warn');
                }
            });

            $('.input_wrap').on('click', '.remove', function(e){
                $(this).parent('div').remove();
                x--;
                $('#counter').html(x);
                $('.warn').removeClass('warn');
                $('#limit').html('Limits not reached');
            })
        });
    </script>
    </body>
</html>
```

Figure 1: A simple HTML5 application (see [6]).

```
.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');
```

The selector `.warn` matches *all*[2] nodes in the tree with class `warn`. The call to `removeClass()` removes the class `warn` from these nodes. Observe that when this call is invoked, the CSS rule above will no longer be matched.

Some jQuery calls may contain an event listener. E.g.

```
$('.button').click(...);
```

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', ...);
```

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></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 descendants 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 not updated to reflect the change (e.g. see [7]). 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 [4]). Some popular web pages are known to have up to 60% redundant CSS rules, as suggested by recent case studies [35].

[2]Unlike node IDs, a single class might be associated with multiple nodes

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 analysis 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 $2^{\mathcal{K}}$ of node labels (where $\mathcal{K}$ 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 [32]) 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 [13].

A tree-rewrite rule σ in our rewrite systems is a tuple (g, χ) consisting of a node selector g (a.k.a. *guard*) and a rewrite operation χ. 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) $('#limit').addClass('warn');
(2) $('.warn').removeClass('warn');
(3) $('.input_wrap').append('<div>
      <input/><a class="remove"></a></div>');
(4) $('.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 data-flow 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 $\mathbb{N} = \{0, 1, \cdots\}$) and a partial order $\leq$ over D satisfying: (1) $\leq$ is the prefix-of relation, i.e., $v \leq w$ iff $w = v.u$ for some $u \in \mathbb{N}^*$, (2) D is prefix-closed, i.e., $w.i \in D$ with $i \in \mathbb{N}$ implies $w \in D$.

An *(labeled) tree* over the nonempty finite set (a.k.a. alphabet) Σ is a logical structure of the form $T = (D, \leq, \lambda)$ where $(D, \leq)$ is a tree structure and λ is a mapping (a.k.a. *node labeling*) from D to Σ. 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 $\textsc{Tree}(\Sigma)$ denote the set of trees over Σ. For every $k \in \mathbb{Z}_{>0}$, we define $\textsc{Tree}_k(\Sigma)$ 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 Trs for HTML5 tree updates. A *rewrite system* $\mathcal{R}$ in Trs is a (finite) set of *rewrite rules*. Each rule σ is a tuple (g, χ) of a guard g and a (rewrite) operation χ. 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 [38, 33, 26]. More formally, a *guard* over the node labeling $\Sigma = 2^{\mathcal{K}}$ with $\mathcal{K} = \{c_1, \cdots, c_n\}$ can be defined by the following grammar:

$$g ::= \top \mid c \mid g \wedge g \mid g \vee g \mid \neg g \mid \langle d \rangle g$$

where c ranges over $\mathcal{K}$ 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 $v, T \models c$ (for a class $c \in \mathcal{K}$) as $c \in \lambda(v)$, and each modality $\langle d \rangle$ (where $d \in \{\uparrow, \uparrow^+, \downarrow, \downarrow^+\}$) in accordance with the arrow orientation (possibly with a transitive closure[3]). See the appendix 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 Σ-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) $\texttt{AddChild}(X)$, (2) $\texttt{AddClass}(X)$, (3) $\texttt{RemoveClass}(X)$, and (4) `RemoveNode`, where $X \subseteq \mathcal{K}$. Intuitively, the semantics of Operations (2)–(4) coincides with the semantics of the jQuery actions `addClass(.)`, `removeClass(.)`, and `remove()`, respectively, while the semantics of $\texttt{AddChild}(X)$ coincides with the semantics of the jQuery action `append(string1)` in the case when `string1` represents a single node associated with classes X. By adding extra classes, appending a larger subtree can be easily simulated by several steps of $\texttt{AddChild}(X)$ 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 σ (written $T \to_\sigma T'$) if there exists a node $v \in D$ such that $v \models g$ and

- if $\chi = \texttt{AddClass}(X)$ then $D' = D$ and $\lambda' := \lambda[v \mapsto X \cup \lambda(v)]$. [4]
- if $\chi = \texttt{AddChild}(X)$ then $D' = D \cup \{v.i\}$ and $\lambda' := \lambda[v.i \mapsto X]$ where i is the number of children of v in T
- if $\chi = \texttt{RemoveClass}(X)$ then $D' = D$ and $\lambda' := \lambda[v \mapsto \lambda(v) \setminus X]$
- if $\chi =$ `RemoveNode` and v is *not* the root node, $D' := D \setminus \{v.w : w \in \mathbb{N}^*\}$ and λ' is the restriction of λ to D'.

Given a rewrite system $\mathcal{R}$ over Σ-trees, we define $\to_{\mathcal{R}}$ to be the union of $\to_\sigma$, for all $\sigma \in \mathcal{R}$. For every $k \in \mathbb{Z}_{>0}$, we define $\to_{\mathcal{R},k}$ to be the restriction of $\to_{\mathcal{R}}$ to $\textsc{Tree}_k(\Sigma)$. Given a set $\mathcal{C}$ of Σ-trees, we write $post^*_{\mathcal{R}}(\mathcal{C})$ (resp. $post^*_{\mathcal{R},k}(\mathcal{C})$) to be the set of trees T' satisfying $T \to^*_{\mathcal{R}} T'$ (resp. $T \to^*_{\mathcal{R},k} T'$) 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 $(\texttt{\#limit}, \texttt{AddClass}(\{\texttt{.warn}\}))$, while the second rule translates to $(\texttt{.warn}, \texttt{RemoveClass}(\{\texttt{.warn}\}))$.

[3] If desired, we can include extra transitive-reflexive closure modalities $\langle\downarrow^*\rangle$ and $\langle\uparrow^*\rangle$ while retaining all upper and lower bound complexity.

[4] Given a map $f : A \to B$, $a' \in A$ and $b' \in B$, we write $f[a' \mapsto b']$ to mean the map $(f \setminus \{(a', f(a'))\}) \cup \{(a', b')\}$

The fourth rule identifies `div` nodes that have some child with class `remove` that in turn has some ancestor with class `input_wrap`. Thus, it translates to

$$(\texttt{div} \wedge \langle\downarrow\rangle(\texttt{.remove} \wedge \langle\uparrow^+\rangle\texttt{.input\_wrap}), \texttt{RemoveNode}).$$

Finally, the third rule requires the construction of a new sub-tree. 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:

$$(\texttt{.input\_wrap}, \texttt{AddChild}(\{\texttt{div}, \texttt{tmp}\}))\ .$$

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

$$\begin{array}{ll} & (\texttt{tmp}, \texttt{AddChild}(\{\texttt{input}\})) \\ \text{and} & (\texttt{tmp}, \texttt{AddChild}(\{\texttt{a}, \texttt{.remove}\}))\ . \end{array}$$

The redundancy problem The *redundancy problem for* TRS is the problem that, given a rewrite system $\mathcal{R}$ 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 $S' \subseteq S$ of guards that are not matched in $post^*_{\mathcal{R}}(T_0)$. The decision version of the redundancy problem for TRS is simply to check if the aforementioned set S' is empty. Similarly, for each $k \in \mathbb{Z}_{>0}$, we define the *k-redundancy problem for* TRS to be the restriction of the redundancy problem for TRS to trees of height k (i.e. we use $post^*_{\mathcal{R},k}$ instead of $post^*_{\mathcal{R}}$).

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 [21]). [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 appendix.

Proposition 1. *The 1-redundancy problem for $\mathcal{R}$ 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 [35] containing 15 live web applications. In practice, negations are almost always limited to negating atomic formulas, i.e., $\neg c$ (for a class $c \in \mathcal{K}$) which can be overapproximated by $\top$ often without losing too much precision[5]. A main result of the paper is that the "monotonic" abstraction that is obtained by restricting to positive guards gives us

[5] In the case when c is an HTML tag name (e.g. `div` or `img`), then we can overapproximate this with the positive formula $\bigvee_{c' \in \mathcal{C} \setminus \{c\}} c'$, where $\mathcal{C}$ is the set of all HTML tag names in the input HTML5 application, without losing any precision

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* $\textsc{Trs}_0$ *the set of rewrite systems with positive guards. The guard databases in the input to redundancy and k-redudancy problems for* $\textsc{Trs}_0$ *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 $R \subseteq S \times S$ is a *preorder* if it is transitive, i.e., if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$. We start with a definition of a preorder $\preceq$ over $\textsc{Tree}(\Sigma)$, where $\Sigma = 2^{\mathcal{K}}$. Given two Σ-trees $T = (D, \lambda)$ and $T' = (D', \lambda')$, we write $T \preceq T'$ if there exists an *embedding* from T' to T, i.e., a function $f : D \to D'$ such that:

(H1) $f(\epsilon) = \epsilon$

(H2) For each $v \in D$, $\lambda(v) \subseteq \lambda'(f(v))$

(H3) For each $v.a \in D$ where $v \in \mathbb{N}^*$ and $a \in \mathbb{N}$, we have $f(v.a) = f(v).b$ for some $b \in \mathbb{N}$.

Note that this is equivalent to the standard notion of *homomorphisms* from database theory (e.g. see [10]) when each class $c \in \mathcal{K}$ is treated as a unary relation. The following is a basic property of $\preceq$, whose proof is easy and is left to the reader.

Fact. $\preceq$ *is a preorder on* $\textsc{Tree}(\Sigma)$.

As an aside, the non-reflexive version $\prec$ of $\preceq$ is a *well-quasi-order* since it does not allow an infinite descending chain $T_1 \succ T_2 \succ \cdots$. The following lemma shows that embeddings preserve positive guards.

Lemma 1. *Given trees* $T = (D, \lambda)$ *and* $T' = (D', \lambda') \in \textsc{Tree}(\Sigma)$ *satisfying* $T \preceq T'$ *with a witnessing embedding* $f : D \to D'$*, and a positive guard* g *over* Σ*, then if* $v, T \models g$ *then* $f(v), T' \models g$.

We relegate to the appendix the proof of Lemma 1, which is similar to (part of) the proof of the homomorphism theorem for conjunctive queries (e.g. see [10, Theorem 6.2.3]). This lemma yields the following monotonicity property of $\textsc{Trs}_0$.

Lemma 2 (Monotonicity)**.** *For each* $\sigma \in \mathcal{R}$*, if* $T_1 \preceq T_2$ *and* $T_1 \to_\sigma T_1'$*, then either* $T_1' \preceq T_2$ *or* $T_2 \to_\sigma T_2'$ *for some* T_2' *satisfying* $T_1' \preceq T_2'$.

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 appendix.

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 formalised in the following lemma, whose proof is given in the appendix.

Lemma 3. *Given a rewrite system* $\mathcal{R}$ *over* Σ*-trees, a guard database* S*, and an initial tree* T_0*, let* $\mathcal{R}^-$ *be the set of* $\mathcal{R}$*-rules less those 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 $\textsc{Trs}_0$. Our first result is:

Theorem 1. *The redundancy problem for* $\textsc{Trs}_0$ *is* EXP*-complete.*

Although obtaining decidability per se is not difficult (e.g. by using Lemma 2 and the theory of well-structured transition systems [8, 17], which only gives a nonelementary complexity), obtaining a tight complexity bound for the redundancy problem for $\textsc{Trs}_0$ 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. Bebop [14], Getafix [24], and Moped [37]). 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 appendix. In the case of k-redundancy problem, a better complexity can be obtained.

Theorem 2. *The k-redundancy problem* $\textsc{Trs}_0$ *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* W[1]*-hard.*

The W[1]-hardness in this theorem is evidence from parameterised complexity theory [18] 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 appendix.

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 $\textsc{Trs}_0$. 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 $\mathcal{P} = (\mathcal{Q}, \Gamma, \Delta)$, where $\mathcal{Q}$ is a finite set of *control states*, Γ is a finite set of *stack symbols*, and Δ 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 $\rightarrow_{\mathcal{P}} \subseteq (\mathcal{Q} \times \Gamma^*) \times (\mathcal{Q} \times \Gamma^*)$ as follows: $(q, v) \rightarrow_{\mathcal{P}} (q', v')$ if there exists a rule $((q, a), (q', w)) \in \Delta$ such that $v = ua$ and $v' = uw$ 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 Δ is a

finite set of pairs (i, φ) of number $i \in \{0, 1, 2\}$ and boolean formula φ 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 $\mathcal{W}' := \bigcup_{j=1}^{i} \mathcal{W}_j$ where $\mathcal{W}_j := \{y_1^j, \ldots, y_m^j\}$. This sPDS generates a (exponentially bigger) pushdown system $\mathcal{P} = (\mathcal{Q}, \Gamma, \Delta')$, where $\mathcal{Q} = \{0,1\}^n$, $\Gamma = \{0,1\}^m$, and $((q, a), (q', w)) \in \Delta'$ iff there exists a pair $(i, \varphi) \in \Delta$ satisfying $i = |w|$, and φ is satisfied by the assignment that assigns[6] q to $\mathcal{V}$, a to $\mathcal{W}$, q' to $\mathcal{V}'$, and w to $\mathcal{W}'$ (i.e. assigning the jth letter of w to $\mathcal{W}_j$). As we will show in the appendix, 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 [39, 37]).

The *bit-toggling problem for sPDS* is a simple reachability problem over symbolic pushdown systems: given an sPDS $\mathcal{P} = (\mathcal{V}, \mathcal{W}, \Delta)$ 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 $I_0 \to_{\mathcal{P}}^* (q, a)$ 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 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 2. *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 [39]), which for completeness we provide in the appendix.

5.2 Intuition/Roadmap of the reduction

The following theorem formalises our reduction claim.

Theorem 3. *The redundancy (resp. k-redundancy) problem for* $\textsc{Trs}_0$ *is polynomial-time reducible to the bit-toggling (resp. bounded bit-toggling) problem for sPDS.*

Together with Proposition 2, Theorem 3 implies an EXP (resp. PSPACE) upper bound for the redundancy (resp. k-redundancy) problem for sPDS. Moreover, as discussed in the appendix, 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 $\mathcal{R} \in \textsc{Trs}_0$, an initial tree $T = (D, \lambda) \in \textsc{Tree}(\Sigma)$ with $\Sigma = 2^{\mathcal{K}}$, 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 $(g, \texttt{AddClass}(X)) \in \mathcal{R}$ at v. The second saturation rule is more involved. We construct a pushdown system $\mathcal{P}$ 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 for each class $c \in \mathcal{K}$. If $\mathcal{P}$ reaches a stack configuration $X \subseteq 2^{\mathcal{K}}$ (i.e. containing only one item) satisfying $X \supset \lambda(v)$, then we set $\lambda(v) := X$. The reason why it suffices to only keep track of $\lambda(v)$ (instead of the entire subtree rooted at v that $\mathcal{P}$ explored) is monotonicity of $\textsc{Trs}_0$ (cf. Lemma 2), i.e., that $\lambda(v)$ contains sufficient information to "regrow" the destroyed subtree. In sum, our "reduction"

[6] Meaning that if $q = (q_1, \ldots, q_n)$, then q_i is assigned to x_i

is in fact an algorithm for the (k-)redundancy problem for $\textsc{Trs}_0$ that runs in polynomial-time with an oracle to the bit-toggling (resp. for bounded stack height) for sPDS.

The actual reduction in Theorem 3 involves several intermediate polynomial-time reductions. Here is a roadmap. We first show that it suffices to consider "simple" rewrite systems 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 $\textsc{Trs}_0$ is similar but each intermediate problem is relativised 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: $S' = \{g \vee \langle\downarrow^+\rangle g : g \in S\}$. Then, for each tree $T = (D, \lambda) \in \textsc{Tree}(\Sigma)$ and guard g, it is the case that $(\exists v \in D : v, T \models g)$ iff $\epsilon, T \models g \vee \langle\downarrow^+\rangle g$.

We now proceed to simplification (2). A guard over the node labeling $\Sigma = 2^{\mathcal{K}}$ 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 $\mathcal{K}$ and d ranges over $\{\uparrow, \downarrow\}$. [Note: if $m = 0$, then $\bigwedge_{i=1}^{m} c_i \equiv \top$.] For notational convenience, if $X = \{c_1, \ldots, c_m\}$, we shall write X (resp. $\langle d\rangle X$) to mean $\bigwedge_{i=1}^{m} c_m$ (resp. $\langle d\rangle \bigwedge_{i=1}^{m} c_m$). A rewrite system $\mathcal{R} \in \textsc{Trs}_0$ is said to be *simple* if (i) all guards occuring in $\mathcal{R}$ are simple, and (ii) if $(\langle d\rangle X, \chi) \in \mathcal{R}$, then χ is of the form $\texttt{AddClass}(Y)$. We define $\textsc{Trs}'_0 \subseteq \textsc{Trs}_0$ to be the set of simple rewrite systems. The redundancy (resp. k-redundancy) problem for $\textsc{Trs}'_0$ is defined in the same way as for $\textsc{Trs}_0$ *except that* all the guards in the guard database are restricted to be a subset of $\mathcal{K}$. The following lemma shows that the redundancy (resp. k-redundancy) problem for $\textsc{Trs}_0$ can be reduced in polynomial time to the redundancy (resp. k-redundancy) problem for $\textsc{Trs}'_0$.

Lemma 4. *Given a $\mathcal{R} \in \textsc{Trs}_0$ over $\Sigma = 2^{\mathcal{K}}$ and a guard database S over Σ, there exists $\mathcal{R}' \in \textsc{Trs}'_0$ over $\Sigma' = 2^{\mathcal{K}'}$ (where $\mathcal{K} \subseteq \mathcal{K}'$) and a set $S' \subseteq \mathcal{K}'$ of simple guards such that:*

(P1) *For each $k \in \mathbb{N}$, S is k-redundant for $\mathcal{R}$ iff S' is k-redundant for $\mathcal{R}'$.*

(P2) *S is redundant for $\mathcal{R}$ iff S' is redundant for $\mathcal{R}'$.*

Moreover, we can compute $\mathcal{R}'$ and S' in polynomial time.

We show how to compute $\mathcal{R}'$. The set $\mathcal{K}'$ is defined as the union of $\mathcal{K}$ with the set G of all subformulas (i.e. occuring in the parse tree) of guard formulas in S and $\mathcal{R}$. In the sequel, to avoid potential confusion, we will often underline members of G in $\mathcal{K}'$, e.g., write $\underline{\langle\downarrow\rangle c}$ instead of $\langle\downarrow\rangle c$.

We now define the simple rewrite system $\mathcal{R}'$. Initially, we will define a rewrite system $\mathcal{R}_1$ that allows the operators $\langle\uparrow^+\rangle$ and $\langle\downarrow^+\rangle$; later we will show how to remove them. We first add the following "intermediate" rules to $\mathcal{R}_1$:

1. $(\{g, g'\}, \texttt{AddClass}(\underline{g \wedge g'}))$, for each $(g \wedge g') \in G$.
2. $(g, \texttt{AddClass}(\underline{g \vee g'}))$ and $(g', \texttt{AddClass}(\underline{g \vee g'}))$, for each $(g \vee g') \in G$.

3. $(\langle d\rangle \underline{g}, \texttt{AddClass}(\underline{\langle d\rangle g}))$, for each $\langle d\rangle g \in G$.

4. $(\underline{g}, \chi)$, for each $(g, \chi) \in \mathcal{R}$.

Note that in Rule (3) the guard $\langle d\rangle \underline{g}$ is understood to mean a non-atomic guard over $2^{\mathcal{K}'}$. Finally, we define $S' := \{\underline{g} : g \in S\}$. Notice that each guard in S' is atomic. The aforementioned algorithm computes $\mathcal{R}_1$ 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 $\mathcal{R}'$. Initially, we set $\mathcal{R}' := \mathcal{R}_1$. Next, for each rule $(\langle d^+\rangle \underline{g}, \chi)$ in $\mathcal{R}_1$ where $d \in \{\uparrow, \downarrow\}$, we add the following rules to $\mathcal{R}'$:

(a) $(\langle d\rangle \underline{g}, \texttt{AddClass}(\underline{\langle d^+\rangle g}))$.

(b) $(\langle d\rangle \underline{\langle d^+\rangle g}, \texttt{AddClass}(\underline{\langle d^+\rangle g}))$.

Note that $\mathcal{R}_1$ contains the rule $(\underline{\langle d^+\rangle g}, \chi)$, where $\underline{\langle d^+\rangle g}$ is understood to mean an atomic guard over $2^{\mathcal{K}'}$. Intuitively, this simplification can be done because $v, T \models \langle d^+\rangle g$ iff at least one of the following cases holds: (i) $v, T \models \langle d\rangle g$, (ii) there exists a node w in T such that $w, T \models \langle d^+\rangle g$ and w can be reached from v by following the direction d for one step. The aforementioned computation step again can be done in linear time. The proof of correctness (i.e. **(P1)** and **(P2)**) is provided in the appendix. In particular, for all $g \in S$, it is the case that g is redundant in $\mathcal{R}$ iff $\underline{g}$ is redundant in $\mathcal{R}'$.

5.4 Redundancy → class-adding

We will show that redundancy for $\textsc{Trs}'_0$ can be solved in polynomial time assuming an oracle to the "class adding problem" for $\textsc{Trs}'_0$. The class adding problem is a reachability problem for $\textsc{Trs}'_0$ 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 $\textsc{Trs}'_0$ lends itself to a fast reduction to the bit-toggling problem for sPDS. 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 $\Sigma = 2^{\mathcal{K}}$ is a function mapping each element of $\{\texttt{root}\} \cup \mathcal{K}$ to $\{0, 1\}$. The boolean value of $f(\texttt{root})$ is used to indicate whether the input single-node tree is a root node. Given a tree $T = (D, \lambda) \in \textsc{Tree}(\Sigma)$, a node $v \in D$, and a simple guard g over Σ, 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) $v = \epsilon$, $g = \langle\uparrow\rangle X$, $f(\texttt{root}) = 0$, and $X \subseteq \{c \in \mathcal{K} : f(c) = 1\}$. In other words, $v, T \models_f g$ checks whether g is satisfied at node v assuming the assumption function f (in particular, if $f(\texttt{root}) = 0$, then any guard referring to the parent of the root node of T is checked against f).

Given a simple rewrite system $\mathcal{R} \in \textsc{Trs}'_0$ over Σ and an assumption function f, we may define the rewriting relation $\to_{\mathcal{R},f} \subseteq \textsc{Tree}(\Sigma) \times \textsc{Tree}(\Sigma)$ in the same way as we define $\to_{\mathcal{R}}$, except that $\models_f$ is used to check guard satisfaction. The *class-adding (reachability) problem* is defined as follows: given a single-node tree $T_0 = (\{\epsilon\}, \lambda_0) \in \textsc{Tree}(\Sigma)$ with $\Sigma = 2^{\mathcal{K}}$, an assumption function $f : (\{\texttt{root}\} \cup \mathcal{K}) \to \{0, 1\}$, a class $c \in \mathcal{K}$, and a simple rewrite system $\mathcal{R}$, decide if there exists a tree $T = (D, \lambda)$ such that $T_0 \to^*_{\mathcal{R},f} T$ 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 5. *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 $T_0 = (D_0, \lambda_0) \in \textsc{Tree}(\Sigma)$ with $\Sigma = 2^{\mathcal{K}}$, a simple rewrite system $\mathcal{R}$ over Σ, and a set $S \subseteq \mathcal{K}$, 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 $(g, \texttt{AddClass}(B))$ 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 $\langle T_v, f, c, \mathcal{R}\rangle$, then $\lambda(v) := \lambda(v) \cup \{c\}$, for some $v \in D$, $c \in \mathcal{K} \setminus \lambda(v)$, and $T_v := (\epsilon, \lambda_v)$ with $\lambda_v(\epsilon) = \lambda(v)$, where we define $f : (\{\texttt{root}\} \cup \mathcal{K}) \to \{0, 1\}$ with $f(\texttt{root}) = 1 \Leftrightarrow v = \epsilon$ and, if $f(\texttt{root}) = 0$ and u is the parent of v, then $f(a) = 1 \Leftrightarrow a \in \lambda(u)$.

Observe that saturation rules can be applied at most $\mathcal{K} \times |D|$ 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 1 and Lemma 2; see the appendix.

5.5 Class-adding → bit-toggling

Lemma 6. *The class-adding (resp. k-class-adding) problem for* $\textsc{Trs}'_0$ *is polynomial-time reducible to the bit-toggling (resp. bounded bit-toggling) problem for sPDS.*

We prove the lemma above. Fix a simple rewrite system $\mathcal{R}$ over the node labeling $\Sigma = 2^{\mathcal{K}}$, an assumption function $f : (\{\texttt{root}\} \cup \mathcal{K}) \to \{0, 1\}$, a single node tree $T_0 = (\{\epsilon\}, \lambda_0) \in \textsc{Tree}(\Sigma)$, 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 $\mathcal{V} := \{x_c : c \in \mathcal{K}\} \cup \{\texttt{pop}\}$, and $\mathcal{W} := \{y_c, z_c : c \in \mathcal{K}\} \cup \{\texttt{root}\}$. Roughly speaking, we will use the variable y_c (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 `root` signifies whether the current node is a root node, while the variable `pop` indicates whether the last operation that changed the stack height is a pop. We next define Δ:

- For each $(A, \texttt{AddClass}(B)) \in \mathcal{R}$, let $C := \mathcal{K} \setminus (A \cup B)$. Now add the rule $(1, \varphi)$, where φ is a conjunction of: (a) $\texttt{pop} \leftrightarrow \texttt{pop}'$, (b) $\texttt{root} \leftrightarrow \texttt{root}^1$, (c) $\bigwedge_{c \in C}(y_c \leftrightarrow y_c^1)$, (d) $\bigwedge_{a \in A}(y_a \wedge y_a^1)$, (e) $\bigwedge_{b \in B \setminus A} y_b^1$, (f) $\bigwedge_{c \in \mathcal{K}}(z_c \leftrightarrow z_c^1)$, and (g) $\bigwedge_{c \in \mathcal{K}}(x_c \leftrightarrow x'_c)$.
- For each $(A, \texttt{AddChild}(B)) \in \mathcal{R}$, add the rule $(2, \varphi)$, where φ is a conjunction of: (a) $\neg\texttt{pop}' \wedge \neg\texttt{root}^2 \wedge (\texttt{root} \leftrightarrow \texttt{root}^1)$, (b) $\bigwedge_{c \in \mathcal{K} \setminus A}(y_c \leftrightarrow y_c^1 \leftrightarrow z_c^2)$, (c) $\bigwedge_{c \in \mathcal{K}}(z_c \leftrightarrow z_c^1)$, (d) $\bigwedge_{a \in A}(y_a \wedge y_a^1 \wedge z_a^2)$, (e) $\bigwedge_{c \in \mathcal{K}} \neg x'_c$, (f) $\bigwedge_{b \in B} y_b^2$, and (g) $\bigwedge_{c \in \mathcal{K} \setminus B} \neg y_c^2$.

- For each $(\langle\uparrow\rangle A, \texttt{AddClass}(B)) \in \mathcal{R}$, add the rule $(1, \varphi)$ where φ is a conjunction of: (a) $\texttt{pop} \leftrightarrow \texttt{pop}'$, (b) $\texttt{root} \leftrightarrow \texttt{root}^1$, (c) $\bigwedge_{c\in\mathcal{K}}(x_c \leftrightarrow x'_c)$, (d) $\bigwedge_{a\in A}(z_a \wedge z^1_a)$, (e) $\bigwedge_{c\in\mathcal{K}\setminus A}(z_c \leftrightarrow z^1_c)$, (f) $\bigwedge_{b\in B} y^1_b$, and (g) $\bigwedge_{c\in\mathcal{K}\setminus B}(y_c \leftrightarrow y^1_c)$.
- For each $(\langle\downarrow\rangle A, \texttt{AddClass}(B)) \in \mathcal{R}$, add the rule $(1, \varphi)$ where φ is a conjunction of: (a) $\texttt{pop} \wedge \texttt{pop}'$, (b) $\texttt{root} \leftrightarrow \texttt{root}^1$, (c) $\bigwedge_{a\in A}(x_a \wedge x'_a)$, (d) $\bigwedge_{c\in\mathcal{K}\setminus A}(x_c \leftrightarrow x'_c)$, (e) $\bigwedge_{c\in\mathcal{K}}(z_c \leftrightarrow z^1_c)$, (f) $\bigwedge_{b\in B} y^1_b$, and (g) $\bigwedge_{c\in\mathcal{K}\setminus B}(y_c \leftrightarrow y^1_c)$.
- Finally, add the rule $(0, \varphi)$, where φ is a conjunction of: (a) $\texttt{pop}' \wedge \neg\texttt{root}$, (b) $\bigwedge_{c\in\mathcal{K}}(y_c \leftrightarrow x'_c)$.

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

Continuing with our translation, the bit that needs to be toggled on is y_α. 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: $I_{X,f,q}(\texttt{root}) := f(\texttt{root})$, $I_{X,f,q}(\texttt{pop}) := q(\texttt{pop})$, and for each $c \in \mathcal{K}$: (i) $I_{X,f,q}(x_c) := q(x_c)$, (ii) $I_{X,f,q}(y_c) = 1$ iff $c \in X$, 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 $I_{\lambda_0(\epsilon),f}$.

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 7. *For each subset $X \subseteq \mathcal{K}$, the following are equivalent:*

(A1) *There exists a tree $T = (D, \lambda) \in \textsc{Tree}(\Sigma)$ such that $T_0 \to^*_{\mathcal{R},f} T$ and $\lambda(\epsilon) = X$.*

(A2) *There exists $q' : \mathcal{V} \to \{0, 1\}$ such that $I_{\lambda_0(\epsilon),f} \to^*_{\mathcal{P}} I_{X,f,q'}$.*

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

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 [41] 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_v, 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 [39] 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 $\langle T_v, f, c, \mathcal{R} \rangle$ 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 appendix.

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 [35], and is described in detail below. The second (and largest) is based on the Nivo Slider plugin [40] 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` 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].

Case Study	Ns	Ss	Ls	Rs	Time
`bikes.html`	23	18 (0)	97	37	3.6s
`comments.html`	5	13 (1)	43	26	2.9s
`example.html`	11	1 (0)	28	4	.6s
`example-up.html`	3	1 (1)	15	3	.6s
`igloo/`		261 (89)			3.4s
`index.html`	145		24	1	
`engineering.html`	236		24	1	
`Nivo-Slider/`					
`demo.html`	15	172 (131)	501	21	6.3s
`transactions.html`	20	9 (0)	37	6	1.6s

Table 1: Case study results.

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.

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[7] 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

```
#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

```
h2 a:hover, h2 a:active, h2 a:focus
h3 a:hover, h3 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 ignore 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

[7]Not including the number of rules required to represent the CSS selectors.

in fact be "`h3 a:focus`". This raises the question of selector subsumption, on which there are already a lot of research works (e.g. see [15, 35]). 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 not 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, given 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

[1] `http://trends.builtwith.com/javascript/jQuery`. jQuery stats (Dec 2014).

[2] `http://www.w3schools.com/`. W3 Schools.

[3] `http://www.sanwebe.com/2013/03/addremove-input-fields-dynamically-with-jquery`.

[4] `http://www.websiteoptimization.com/speed/tweak/average-web-page/`. Average Web Page Size.

[5] `https://bitbucket.org/matthewhague/treeped`.

[6] `http://homepages.inf.ed.ac.uk/v1awidja/box.html`, Dec 2014. A simple HTML5 application.

[7] `http://homepages.inf.ed.ac.uk/v1awidja/box1.html`, Dec 2014. A no-limit version of [6].

[8] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

[9] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In *CAV*, pages 555–568, 2002.

[10] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1994.

[11] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active XML. In *PODS*, pages 35–45, 2004.

[12] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active XML systems. *ACM Trans. Database Syst.*, 34(4), 2009.

[13] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for XML. In *WWW*, pages 486–495, 2002.

[14] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[15] Martí Bosch, Pierre Genevès, and Nabil Layaïda. Automated refactoring for size reduction of CSS style sheets. In *DocEng*, pages 13–16, 2014.

[16] Wenfei Fan, Floris Geerts, and Frank Neven. Expressiveness and complexity of XML publishing transducers. *ACM Trans. Database Syst.*, 33(4), 2008.

[17] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[18] J. Flum and M. Grohe. *Parameterized Complexity Theory.* Springer, 2006.

[19] Blaise Genest, Anca Muscholl, Olivier Serre, and Marc Zeitoun. Tree pattern rewriting systems. In *ATVA*, pages 332–346, 2008.

[20] Blaise Genest, Anca Muscholl, and Zhilin Wu. Verifying recursive active documents with positive data tree rewriting. In *FSTTCS*, pages 469–480, 2010.

[21] Pierre Geneves, Nabil Layaida, and Vincent Quint. On the Analysis of Cascading Style Sheets. In *WWW*, pages 809–818, 2012.

[22] Matthew Hague. Senescent ground tree rewrite systems. In *CSL-LICS*, page 48, 2014.

[23] jQuery website. `http://jquery.com/`.

[24] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI*, pages 211–222, 2009.

[25] Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. Combining form and function: Static types for jquery programs. In *ECOOP*, pages 79–103, 2013.

[26] Leonid Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[27] Anthony Widjaja Lin. Accelerating tree-automatic relations. In *FSTTCS*, pages 313–324, 2012.

[28] Anthony Widjaja Lin. Weakly-synchronized ground tree rewriting. In *MFCS*, pages 630–642, 2012.

[29] Christof Löding. Reachability problems on regular ground tree rewriting graphs. *Theory Comput. Syst.*, 39(2):347–383, 2006.

[30] Christof Löding and Alex Spelten. Transition graphs of rewriting systems over unranked trees. In *MFCS*, pages 67–77, 2007.

[31] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML type checking with macro tree transducers. In *PODS*, pages 283–294, 2005.

[32] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, 1995.

[33] Maarten Marx. Conditional xpath. *ACM Trans. Database Syst.*, 30(4):929–959, 2005.

[34] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *FSE*, pages 496–506, 2014.

[35] Ali Mesbah and Shabnam Mirshokraie. Automated Analysis of CSS Rules to Support Style Maintenance. In *ICSE*, pages 408–418, 2012.

[36] Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *WWW*, pages 711–720, 2010.

[37] Moped. `http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`.

[38] Bernd-Holger Schlingloff. Expressive completeness of temporal logic of trees. *Journal of Applied Non-Classical Logics*, 2(2):157–180, 1992.

[39] Stefan Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Technischen Universität München, 2002.

[40] Nivo Slider. `https://github.com/gilbitron/Nivo-Slider`.

[41] Dejvuth Suwimonteerabuth, Felix Berger, Stefan Schwoon, and Javier Esparza. jMoped: A test environment for Java programs. In *CAV*, pages 164–167, 2007.

A Semantics of guard language

Here we define the semantics of the guard language from Section 3. Given a tree $T = (D, \lambda)$ and a node $v \in D$, we define whether v of T *matches* a guard g (written $v, T \models g$) by induction

over the following rules:

- $v, T \models \top$.
- for $c \in \mathcal{K}$, $v, T \models c$ if $c \in \lambda(v)$.
- $v, T \models g \wedge g'$ if $v, T \models g$ and $v, T \models g'$.
- $v, T \models g \vee g'$ if $v, T \models g$ or $v, T \models g'$.
- $v, T \models \neg g$ if it is not the case that $v, T \models g$.
- $v, T \models \langle\uparrow\rangle g$ if $v = w.i$ (for some $i \in \mathbb{N}$), and $w, T \models g$.
- $v, T \models \langle\uparrow^+\rangle g$ if $v = w.w'$ (for some $w' \in \mathbb{N}^+$), and $w, T \models g$.
- $v, T \models \langle\downarrow\rangle g$ if there exists a node $v.i \in D$ (for some $i \in \mathbb{N}$) such that $v.i, T \models g$.
- $v, T \models \langle\downarrow^+\rangle g$ if there exists a node $v.w \in D$ (for some $w \in \mathbb{N}^+$) such that $v.w, T \models g$.

B Proof of Proposition 1

We prove the undecidability of the 1-redundancy problem when negation is allowed in the guards. The proof is by a reduction from the undecidable control-state reachability problem for deterministic two counter machines (2-CM): given a 2-CM $\mathcal{M} = (\mathcal{Q}, \Delta, q_0, q_F)$ with counters X and Y, decide whether there exist a path from $(q_0, 0, 0)$ to some configuration in $\{q_F\} \times \mathbb{N} \times \mathbb{N}$. We will use X and Y to denote the two counters, and Z to range over $\{X, Y\}$. The two counter tests will be $=$ and $>$ and we use $\ominus$ to range over these tests. We may assume that

(i) counter tests and counter increments / decrements never take place in the same transition, i.e., rules are of form $(q, [Z \ominus 0]) \to q'$ or $q \to (q', \mathsf{inc}(Z))$ or $q \to (q', \mathsf{dec}(Z))$, and

(ii) there are no self-loops in the counter machine i.e. there is no transition of the form $(q, \cdot) \to q$.

The initial tree $T_0 = (D_0, \lambda_0)$ is defined to be $D_0 = \{\epsilon\}$ and $\lambda_0(\epsilon) = q_0$. We now construct a rewrite system $\mathcal{R} \in$ TRS. The state of the machine is always recorded in the root node. The values of the two counters X and Y will be encoded by the number of nodes at level 1 (i.e. children of the root node) that are associated with class X and Y, respectively. More precisely, for each transition rule σ of the machine, we introduce a class σ. In addition, we use the following four classes: X, Y, *new*, and *del*. The main problem that we have to overcome in our reduction is to simulate one transition of $\mathcal{M}$ by several simpler steps of T_0 owing to the simplicity of rewrite operations that we allow in TRS. This is in fact the reason for adding the last two classes *new* and *del*, which are just "flags" to indicate whether a node (at level 1) is "new" or is "scheduled to be deleted". Our rewrite system $\mathcal{R}$ operates over $\Sigma := 2^{\mathcal{K}}$ with $\mathcal{K} = \mathcal{Q} \cup \Delta \cup \{X, Y, new, del\}$. Next we introduce the guards: (a) $g(Z = 0) := \neg\langle\downarrow\rangle Z$, (b) $g(Z > 0) := \langle\downarrow\rangle Z$, (c) $\theta_1 := \neg\langle\downarrow\rangle new \wedge \neg\langle\downarrow\rangle del$, and (d) $\theta_2 := \bigwedge_{\gamma \in \Delta} \neg\gamma$. As we shall see, the guard $\theta_1 \wedge \theta_2$ means that there is no "pending job". The rewrite rules for $\mathcal{R}$ are as follows:

- If $\sigma = (q, [Z \ominus 0]) \to q'$, then add the following rules:

  (1) $(q \wedge g(Z \ominus 0) \wedge \theta_1 \wedge \theta_2, \texttt{AddClass}(\{\sigma, q'\}))$

  (2) $(\sigma, \texttt{RemoveClass}(\{q, \sigma\}))$

- If $\sigma = q \to (q', \mathsf{inc}(Z))$, then add:
  (1) $(q \wedge \theta_1 \wedge \theta_2, \texttt{AddClass}(\{\sigma, q'\})$
  (2) $(\sigma \wedge \theta_1, \texttt{AddChild}(\{Z, new\}))$
  (3) $(\sigma \wedge \langle\downarrow\rangle(Z \wedge new), \texttt{RemoveClass}(\{\sigma, q\}))$.
  (4) $(new \wedge \langle\uparrow\rangle\theta_2, \texttt{RemoveClass}(\{new\}))$
- If $\sigma = q \to (q', \mathsf{dec}(Z))$, then add:
  (1) $(q \wedge \theta_1 \wedge \theta_2, \texttt{AddClass}(\{\sigma, q'\}))$
  (2) $(Z \wedge \langle\uparrow\rangle(\sigma \wedge \neg\langle\downarrow\rangle del), \texttt{AddClass}(\{del\}))$
  (3) $(\sigma \wedge \langle\downarrow\rangle(Z \wedge del), \texttt{RemoveClass}(\sigma, q))$
  (4) $(del \wedge \langle\uparrow\rangle\theta_2, \texttt{RemoveClass}(\{Z, del\}))$

Observe that the trees reachable from the initial tree is of height at most 1. Possible labels of the root are only $q \in \mathcal{Q}$ and $\sigma \in \Delta$. Possible labels of a leaf node (necessarily a child of the root) are X, Y, new and del.

It is not difficult to see that $\mathcal{R}$ when initialised in T_0 performs a faithful simulation of the counter machine $\mathcal{M}$ from the configuration $(q, 0, 0)$. Our rewrite system $\mathcal{R}$ is defined in such a way that each group of rewrite rules introduced in $\mathcal{R}$ to simulate a transition σ of $\mathcal{M}$ has to be executed before the next transition of $\mathcal{M}$ can be simulated by $\mathcal{R}$. In particular, to simulate a $\mathcal{M}$ transition σ from a given state $q \in \mathcal{Q}$, our rewrite system $\mathcal{R}$ needs to add σ to the root node, which can only be done if the following properties are satisfied: (1) the label of the root node is $\{q\}$ for some $q \in \mathcal{Q}$, and (2) no child of the root has del or new in the label. This is owing to the conjuncts $\theta_1 \wedge \theta_2$ in the guard. In particular, this means that the root contains *at most* one σ. In addition, if σ is a class of the form $q \to (q', \mathsf{inc}(Z))$ or $q \to (q', \mathsf{dec}(Z))$, then σ can only be removed only after proper simulation of counter increments/decrements has been performed.

In summary, we have that $\mathcal{M}$ reaches the control state q_F iff q_F is not redundant in $\mathcal{R}$. This concludes our reduction.

C Proof of Lemma 1

We prove that if $T \preceq T'$, then T' satisfies at least the same guards as T at each node v. The proof is by induction on g. For the base cases, $g = \top$ is trivial, and that $v, T \models X$ implies $f(v), T' \models X$ holds since, by **(H2)**, we have $X \subseteq \lambda(v) \subseteq \lambda'(f(v))$. We now proceed to the inductive case. The cases of conjunction and disjunction are obvious.

Assume that $v, T \models \langle d\rangle g$ for some $d \in \{\uparrow, \downarrow\}$. We will only show the case when $d = \downarrow$; the other case can be handled in the same way. This means that $v.i, T \models g$ for some $i \in \mathbb{N}$. By induction, we have $f(v.i), T' \models g$. By **(H3)**, we have that $f(v.i) = f(v).j$ for some $j \in \mathbb{N}$. Altogether, this implies that $f(v), T' \models \langle\downarrow\rangle g$ as desired.

Assume that $v, T \models \langle d^+\rangle g$ for some $d \in \{\uparrow^+, \downarrow^+\}$. We will only show the case when $d = \downarrow^+$; the other case can be handled in the same way. The proof is by induction on the length n of path from v to a descendant vw (with $w \neq \epsilon$) satisfying $vw, T \models g$. If $n = 1$, then by the previous paragraph we have $f(v), T' \models \langle\downarrow\rangle g$ and so $f(v), T' \models \langle\downarrow^+\rangle g$ as desired. If $n > 1$ and $w = iu'$ for some $i \in \mathbb{N}$ and $u' \in \mathbb{N}^*$, then $vi, T \models \langle\downarrow^+\rangle g$ with a witnessing path of length $n - 1$. Therefore, by induction, we have $f(vi), T' \models \langle\downarrow^+\rangle g$. Since $f(vi) = f(v).j$ for some $j \in \mathbb{N}$, it follows that $f(v), T' \models \langle\downarrow\rangle\langle\downarrow^+\rangle g$ and so $f(v), T' \models \langle\downarrow^+\rangle g$ as desired.

D Proof of Lemma 2

We prove that, whenever $T_1 \preceq T_2$ and $T_1 \to_\sigma T_1'$, then either $T_1' \preceq T_2$ or $T_2 \to_\sigma T_2'$ with $T_1' \preceq T_2'$.

Suppose $T_1 \to_\sigma T_2$ where $T_1 = (D_1, \lambda_1)$, $T_1' = (D_1', \lambda_1')$ and $\sigma = (g, \chi)$; and $T_1 \preceq T_2 = (D_2, \lambda_2)$ is witnessed by an embedding $f : D_1 \to D_2$. Assume $v, T_1 \vDash g$ for some $v \in D_1$. Then $f(v), T_2 \vDash g$ by Lemma 1. We consider each of the four cases of χ in turn.

Case: $\chi = \texttt{AddClass}(X)$. Then set $T_2' = (D_2', \lambda_2')$ with $D_2' := D_2$ and $\lambda_2' := \lambda_2[f(v) \mapsto \lambda_2(f(v)) \cup X)]$. Hence $T_2 \to_\sigma T_2'$. Moreover we have $T_1' \preceq T_2'$, as witnessed by $f : D_1' \to D_2'$ (note that $D_1' = D_1$ and $D_2' = D_2$).

Case: $\chi = \texttt{AddChild}(X)$. Assume $D_1' = D_1 \cup \{v.i\}$ and $\lambda_1' = \lambda_1[v.i \mapsto X]$ where i is the number of children of v in T_1. Let i' be the number of children of $f(v)$ in T_2. Then set $T_2' = (D_2', \lambda_2')$ with $D_2' := D_2 \cup \{f(v).i'\}$ and $\lambda_2' := \lambda_2[f(v).i' \mapsto X]$. Then $T_2 \to_\sigma T_2'$ and $T_1' \preceq T_2'$ is witnessed by $f' : D_1' \to D_2'$ such that $f' := f[v.i \mapsto f(v).i']$.

Case: $\chi = \texttt{RemoveClass}(X)$ or $\chi = \texttt{RemoveNode}$. In both cases, the function f (restricted to D_1') is still an embedding from T_1' to T_2 and so $T_1' \preceq T_2$.

E Proof of Lemma 3

We show that, when guards are monotonic, the $\texttt{RemoveClass}(B)$ and $\texttt{RemoveNode}$ operations do not affect the redundancy of the guards. The $\Leftarrow$-direction is trivial. For the other direction, assume $T_0 \to_{\sigma_0} T_1 \to_{\sigma_1} \cdots \to_{\sigma_{n-1}} T_n \to_{\sigma_n} T_{n+1} = (D, \lambda)$ where each $\sigma_i = (g_i, \chi_i)$, and $v, T_{n+1} \vDash g$ for some $v \in D$. It suffices to construct trees $T_0', \cdots, T_{n+1}'$ with $T_0' = T_0$ such that g matches T_{n+1}', and for each $i \in [1..n]$, $T_i \preceq T_i'$ and $T_i' \to_{\sigma_i'} T_{i+1}'$ where each σ_i' is *not* a $\texttt{RemoveNode}$ or a $\texttt{RemoveClass}(X)$ operation but may be the identity operation. We shall construct such a sequence of trees by induction on i. The base case follows immediately from the assumptions. Let $i \geq 0$. Suppose T_i' has been constructed such that $T_i \preceq T_i'$. There are two cases. If $\chi_i = \texttt{RemoveNode}$ or $\chi_i = \texttt{RemoveClass}(X)$ then set $T_{i+1}' := T_i'$ and σ_i' is the identity operation; otherwise, thanks to Lemma 2, set $\sigma_i' := \sigma_i$ and we can construct T_{i+1}' such that $T_i' \to_{\sigma_i'} T_{i+1}'$ and $T_{i+1} \preceq T_{i+1}'$. Finally, since $T_{n+1} \preceq T_{n+1}'$ and g matches T_{n+1}, we have g also matches T_{n+1}' by Lemma 1 as required.

F Proof of Proposition 2

To show EXP membership for the bit-toggling problem, we simply compute an exponential-sized PDS from a given sPDS and apply the standard polynomial-time algorithm for solving control-state reachability for PDS (this is the same as emptiness for pushdown automata over a unary input alphabet $\{a\}$).

To show PSPACE membership for the bounded bit-toggling problem, suppose that the input to the problem is $\mathcal{P} = (\mathcal{V}, \mathcal{W}, \Delta)$ and $h \in \mathbb{N}$ with $\mathcal{V} = \{x_1, \ldots, x_n\}$ and $\mathcal{W} = \{y_1, \ldots, y_m\}$. Each configuration is a pair (q, w) of $q \in \{0,1\}^n$ and w is a word over the stack alphabet $\Gamma = \{0,1\}^m$. Since the stack height is always bounded by h, each configuration is of size at most $n + (h \times m)$. Also, checking whether $\sigma = ((q, a), (q', w))$ is a transition rule of $\mathcal{P}$ — where $q, q' \in \{0,1\}^n$, $a \in \{0,1\}^m$, and w is a word of length at most 2 over Γ — can be checked in linear time (and therefore polynomial space) by simply evaluating each symbolic rule of the form $(|w|, \varphi)$ with respect to the boolean assignment σ. [Evaluating BDDs, boolean formulas, and boolean circuits can all be done in linear time.] So, we can obtain a nondeterministic polynomial space

algorithm (and therefore membership in PSPACE) by guessing the symbolic rule that needs to be applied at any given moment.

G Proof of Correctness of Lemma 4

We show correctness of the reduction to simple rewrite systems. To show **(P1)** and **(P2)**, it suffices to show the following four statements:

(P1a) For each $k \in \mathbb{N}$, S is k-redundant for $\mathcal{R}$ iff S' is k-redundant for $\mathcal{R}_1$.

(P1b) For each $k \in \mathbb{N}$, S is k-redundant for $\mathcal{R}_1$ iff S' is k-redundant for $\mathcal{R}'$.

(P2a) S is redundant for $\mathcal{R}$ iff S' is redundant for $\mathcal{R}_1$.

(P2a) S is redundant for $\mathcal{R}$ iff S_1 is redundant for $\mathcal{R}'$.

We first show **(P1a)** and **(P2a)**. To this end, we say that a tree $T = (D, \lambda) \in \textsc{Tree}(\Sigma')$ is *consistent* if, for each $g \in G$ and each node $v \in D$ with $\underline{g} \in \lambda(v)$, it is the case that $v, T \models g$. We say that T is *maximally consistent* if it is consistent and that, for each $g \in G$ and each node $v \in D$, if $v, T \models g$ then $\underline{g} \in \lambda(v)$. Observe that each tree in $\textsc{Tree}(\Sigma)$ (in particular, the initial tree) is consistent and that each of the rules (1–3) preserves consistency. Therefore, for all trees $T \in \textsc{Tree}(\Sigma)$ and $T' \in \textsc{Tree}(\Sigma')$, if $T \to^*_{\mathcal{R}_1} T'$ then T' is consistent. In addition, for a tree $T \in \textsc{Tree}(\Sigma')$, we write $T \cap \Sigma$ to denote the tree that is obtained from T by restricting to node labels from Σ. Let $\Theta \subseteq \mathcal{R}_1$ denote the set of intermediate rules added above. Since the class G is closed under taking subformulas, for each tree $T \in \textsc{Tree}(\Sigma)$, it is the case that $T \to^*_\Theta T'$ for some maximally consistent tree $T' \in \textsc{Tree}(\Sigma')$ satisfying $T' \cap \Sigma = T$. Now, by a simple induction on the length of paths, it follows that that $T_1 = (D_1, \lambda_1) \to^*_{\mathcal{R}} T_2 = (D_2, \lambda_2)$ iff $T_1 \to^*_{\mathcal{R}_1} T_2'$ for some maximally consistent tree T_2' such that $T_2' \cap \Sigma = T_2$. Since S' contains precisely all $\underline{g}$ for which $g \in S$, **(P1a)** and **(P2a)** immediately follow.

We now show **(P1b)** and **(P2b)**. Observe that both rules (a) and (b) preserve tree-consistency since $v, T \models \langle d^+ \rangle g$ iff at least one of the following cases holds: (i) $v, T \models \langle d \rangle g$, (ii) there exists a node w in T such that $w, T \models \langle d^+ \rangle g$ and w can be reached from v by following the direction d for one step. As before, by a simple induction on the length of paths, for all consistent trees $T_1, T_2 \in \textsc{Tree}(\Sigma')$, it is the case that $T_1 \to^*_{\mathcal{R}'} T_2$ iff $T_1 \to^*_{\mathcal{R}_1} T_2$. This implies implies **(P1b)** and **(P2b)**, i.e., the correctness of our entire construction.

Note, in particular, for all $g \in S$, it is the case that g is redundant in $\mathcal{R}$ iff $\underline{g}$ is redundant in $\mathcal{R}'$.

H Proof of Algorithm for Lemma 5

We argue the correctness of the fixpoint algorithm in the proof of Lemma 5, which shows that the redundancy (resp. k-redundancy) problem is polynomial time solvable assuming oracle calls to the class-adding (resp. k-class-adding) problem.

Soundness The proof is by induction over the number of applications of the fixpoint rules. Fix some sequence of rule applications reaching a fixpoint and let $T_i = (D_0, \lambda_i)$ be the tree obtained after ith application. We prove by induction that we have a tree T such that $T_0 \to^*_{\mathcal{R}} T = (D', \lambda')$ and $T_i \preceq T$ and thus our algorithm is sound.

In the base case, when $i = 0$, $T = T_0$ and the proof is trivial. Hence, assume by induction we have a tree T such that $T_0 \to^*_{\mathcal{R},f} T$ and $T_i \preceq T$. There are two cases.

- When we apply a rule $\sigma = (g, \texttt{AddClass}(B))$ at a node $v \in D_0$, then $T_i \to_\sigma T_{i+1}$ and by Lemma 2 (Monotonicity) we obtain T' satisfying the induction hypothesis.
- When the class-adding problem has a positive answer on input $\langle T_v, f, c, \mathcal{R}\rangle$ and we update $\lambda(v) := \lambda(v) \cup \{c\}$, then there is a sequence of rule applications $\sigma_1, \ldots, \sigma_n$ witnessing the positive solution to the class adding problems. By n applications of Lemma 2 we easily obtain T' satisfying the induction hypothesis as required.

Completeness Take any sequence of rules $\sigma_1, \ldots, \sigma_n$ such that

$$T_0 \to_{\sigma_1} T_1 \to_{\sigma_2} \cdots \to_{\sigma_n} T_n$$

where for each i, $T_i = (D_i, \lambda_i)$. Note $D_0 \subseteq D_i$ for each i. Let $T_F = (D_0, \lambda)$ be the tree obtained as the conclusion of the fixpoint calculation. We show $\lambda_i(v) \subseteq \lambda(v)$ for all $v \in D_0$ and hence our algorithm is complete.

We induct over i. In the base case $i = 0$ and the result is trivial. For the induction, if σ_i was applied to $v \notin D_0$ or σ_i is an $\texttt{AddChild}(B)$ rule, the result is also trivial. Else σ_i is of the form $\sigma = (g, \texttt{AddClass}(B))$ and applied to $v \in D_0$. There are several cases.

- When $g = A$ let $v' = v$ and when $g = \langle\uparrow\rangle A$ let v' be the parent of v. Since $v \in D_0$ we also have $v' \in D_0$ and thus $\lambda_i(v') \subseteq \lambda(v)$ and thus σ is applicable at v' and the first fixpoint rule applies. Since T_F is a fixpoint, we necessarily have $B \subseteq \lambda(v)$.
- When $g = \langle\downarrow\rangle A$ there are two cases.
  - If the guard was satisfied by matching with $v' \in D_0$, then we have $B \subseteq \lambda(v)$ exactly as above.
  - Otherwise, the guard was satisfied in the run to T' by matching with $v' \notin D_0$. Thus we can pick out the sub-sequence of $\sigma_1, \ldots, \sigma_n$ beginning with the $\texttt{AddChild}(C)$ rule (applied at v) that created v' and all rules applied at v' or a descendent. By Lemma 2 (Monotonicity) and the fact that all guards are simple, this sequence is a witness to the positive solution of the class-adding problem $\langle T_v, f, c, \mathcal{R}\rangle$ for all $c \in B$ and thus $B \subseteq \lambda(v)$ as required.

I BDD representation in the Proof of Lemma 6

We first recall the definition of BDDs. A *binary decision diagram (BDD)* is a symbolic representation of boolean functions. A *BDD over the variables* $\mathcal{V} = \{x_1, \ldots, x_n\}$ is a rooted directed acyclic graph $G = (V, E)$ together with a mapping $\lambda : (V \cup E) \to (\mathcal{V} \cup \{0, 1\})$ such that:

(i) if $v \in V$ is an internal node, then $\lambda(v) \in \mathcal{V}$ and it has *precisely two* outgoing edges e_1, e_2 labeled by $\lambda(e_1) = 0$ and $\lambda(e_2) = 1$, respectively, and

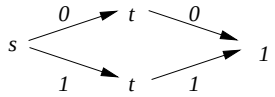
(ii) (ii) if $v \in V$ is a leaf node, then it is labeled by $\lambda(v) \in \{0, 1\}$.

Given an assignment $\nu : \mathcal{V} \to \{0, 1\}$, we can determine the truth value of G under ν as follows: start at the root node of G; if the current node is v labeled by $x \in \mathcal{V}$, follow the outgoing $\nu(x)$-labeled from v to determine the next node; and if the current node is a leaf, then the truth value is given by the label of this leaf.

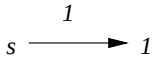
An *ordered BDD (oBDD)* is a BDD together with an ordering of $\mathcal{V}$ (say $x_1 < \cdots < x_n$) such that if v is a predecessor of u in G where both v and u are internal nodes, then $\lambda(v) < \lambda(u)$.

So, no variable $x \in \mathcal{V}$ occurs more than once in each path from the root to a leaf in an oBDD. Finally, an oBDD is reduced if (i) it has no two isomorphic subgraphs, and (ii) it has no internal node whose two children are the same node. In the sequel, we use the term BDD to mean reduced oBDD.

We now demonstrate how to represent boolean formulas in the reduction as BDDs. Observe that each boolean formula in our $\mathcal{P}$-rules is a conjunction of formulas, in which *no two conjuncts share any common variable*. This allows an easy conversion to BDD representation. For example, the BDD $G = (V, E, \lambda)$ for the formula φ in the first type of rules in our sPDS construction in Section 5 can be defined as follows. Each conjunct of the form $s \leftrightarrow t$ is turned into the following BDD



where a missing edge means that it goes to a 0-labeled leaf (also omitted from diagram). Similarly, a conjunct of the form s is turned into the following BDD



where, again, a missing edge means that it goes to a 0-labeled leaf. Now, if G_1 and G_2 are BDDs for the conjuncts φ_1 and φ_2 with different sets of variables, then the BDD for $\varphi_1 \wedge \varphi_2$ can be constructed by gluing the 1-labeled leaf in G_1 by the root of G_2 (using the label of the root of G_2). Since no two conjuncts in φ share a common variable, the BDD for φ can be obtained by linking these simpler BDDs in this way. The resulting BDD is of size linear in the size of φ.

J Proof of Lemma 7

We first prove the direction (A2)⇒(A1). We will prove this by induction on the length of paths. The problem is that our induction hypothesis is not strong enough to get us off the ground (since the state component of $I_{\lambda_0(\epsilon),f}$ is always $(0, \ldots, 0)$). Therefore, we will first prove this by induction with a stronger induction hypothesis:

Lemma 8. *Given $\mathcal{P}$-configurations (q, α) and (q', α') with $q, q' : \mathcal{V} \to \{0,1\}$ and $\alpha, \alpha' : \mathcal{W} \to \{0,1\}$ (i.e. stack of height 1), suppose that $(q, \alpha) \to_{\mathcal{P}}^* (q', \alpha')$. Define $T = (D, \lambda) \in \textsc{Tree}(\Sigma)$ as (1) $D = \{\epsilon\} \cup \{0 : q(\texttt{pop}) = 1\}$, (2) $\lambda(\epsilon) = \{c \in \mathcal{K} : \alpha(y_c) = 1\}$ and $\lambda(0) = \{c \in \mathcal{K} : q(x_c) = 1\}$. Define the assumption function $f_\alpha : (\mathcal{K} \cup \{\texttt{root}\}) \to \{0,1\}$ with $f_\alpha(c) = \alpha(z_c)$ for $c \in \mathcal{K}$ and $f_\alpha(\texttt{root}) = \alpha(\texttt{root})$. Then, there exists $T' = (D', \lambda') \in \textsc{Tree}(\Sigma)$ such that $\lambda'(\epsilon) = \{c \in \mathcal{K} : \alpha'(y_c) = 1\}$ and $T \to_{\mathcal{R}, f_\alpha}^* T'$.*

Observe that the direction (A2)⇒(A1) is immediate from this Lemma: simply apply it with $(q, \alpha) = I_{\lambda_0(\epsilon),f}$ and $(q', \alpha') = I_{X,f,q'}$.

Proof. The proof is by induction on the length k of the witnessing path $\pi : (q, \alpha) \to^* (q', \alpha')$. When $k = 0$, we have that $(q, \alpha) = (q', \alpha')$. So, we may set $T' = T$ and the statement is satisfied.

Let us now consider the case when $k > 0$. There are two cases to consider depending on the second configuration in the path π:

(i) $\pi = (q, \alpha) \to (q_1, \alpha_1) \to^* (q', \alpha')$, for some $q_1 : \mathcal{V} \to \{0,1\}$ and $\alpha_1 : \mathcal{W} \to \{0,1\}$.

(ii) $\pi = (q, \alpha) \to (q_1, \alpha_1\beta_1) \to^* (q', \alpha')$, where $q_1 : \mathcal{V} \to \{0,1\}$ and $\alpha_1, \beta_1 : \mathcal{W} \to \{0,1\}$.

Note that the case when the first action is a pop (i.e. $(q, \alpha) \to (q_1, \epsilon)$) is not possible since (q_1, ϵ) is a deadend.

Let us consider Case (i). The configuration (q_1, α_1) is obtained by applying a rule $\sigma = (1, \varphi) \in \Delta$, which was generated by a rule $\gamma = (g, \texttt{AddClass}(B)) \in \mathcal{R}$. Suppose that $g = A \subseteq \mathcal{K}$; the other two cases (i.e. when g is of the form $(\langle d \rangle A, \texttt{AddClass}(B))$) can be handled in the same way. Since φ has a conjunct of the form y_a for each $a \in A$, it is the case that $A \subseteq \lambda(\epsilon)$. Since y_b^1 is a conjunct of φ for each $b \in B$, The rule σ toggles on the variables y_b for each $b \in B$ while preserving the value of all other variables in $\mathcal{V} \cup \mathcal{W}$. So, if $T_1 := (D, \lambda_1)$ with $\lambda_1(0) := \lambda(0)$ and $\lambda_1(\epsilon) := \lambda(\epsilon) \cup B$, then $T_0 \to_{\mathcal{R}, f_\alpha} T_1$. We may now apply the induction hypothesis on the subpath $(q_1, \alpha_1) \to^* (q', \alpha')$ of π of shorter length and obtain $T_1 \to_{\mathcal{R}, f_\alpha} T'$. Concatenating paths, we obtain a path $T \to_{\mathcal{R}, f_\alpha} T'$ as desired.

Let us now consider Case (ii). The configuration $(q_1, \alpha_1\beta_1)$ is obtained by applying a rule $\sigma = (2, \varphi) \in \Delta$, which was generated by a rewrite rule $(A, \texttt{AddChild}(B)) \in \mathcal{R}$. Therefore, it must be the case that $(q_1, \beta_1) \to^*_{\mathcal{P}} (q_2, \beta_2) \to_{\sigma_2} (q_3, \epsilon)$ and $(q_3, \alpha_1) \to^*_{\mathcal{P}} (q', \alpha')$ for some configurations (q_2, β_2) and (q_3, ϵ) with stack height 1 and 0, respectively, and some $\mathcal{P}$-rule $(0, \varphi')$. As in the previous case, we have $A \subseteq \lambda(\epsilon)$ and so, applying $(A, \texttt{AddChild}(B))$ at ϵ, we obtain $T_1 = (D_1, \lambda_1)$ with $D_1 := D \cup \{1\}$ and λ_1 is an extension of λ to D_1 with $\lambda_1(1) := B$. Let $T_1' = (\{\epsilon\}, \lambda_1')$ be the subtree of T_1 rooted at node 1. Applying induction on the path $(q_1, \beta_1) \to^*_{\mathcal{P}} (q_2, \beta_2)$, we obtain a tree $T_2' = (D_2', \lambda_2')$ such that: (1) $\nu : T_1' \to^*_{\mathcal{R}, f} T_2'$ with assumption function $f := f_{\beta_1}$, and (2) $\lambda_2'(\epsilon) = \{c \in \mathcal{K} : \beta_2(y_c) = 1\}$. Similarly, if we let $T_3 = (\{\epsilon, 0\}, \lambda_3)$ be the tree with $\lambda_3(\epsilon) = \lambda(\epsilon)$ and $\lambda_3(0) = \lambda_2'(\epsilon)$, we apply induction hypothesis on the path $(q_3, \alpha_1) \to^*_{\mathcal{P}} (q', \alpha')$ and obtain a path $\mu : T_3 \to^*_{\mathcal{R}, f_\alpha} T''$ (here $f_\alpha = f_{\alpha_1}$) for some tree $T'' = (D'', \lambda'')$ with $\lambda''(\epsilon) = \{c \in \mathcal{K} : \alpha'(y_c) = 1\}$. Now let T' be the tree obtained from T'' by: (1) removing the node 0 of T'' and replacing it by the tree T_2' (note: $T''(0) = T_2'(\epsilon)$), and (2) adding a node labeled by $\lambda(0)$ as a child of the root node of T''. We have $T'(\epsilon) = T''(\epsilon) = \{c \in \mathcal{K} : \alpha'(y_c) = 1\}$. Finally, we see that $T \to_{\mathcal{R}, f_\alpha} T_1 \to^*_{\mathcal{R}, f_\alpha} T'$, where the path $T_1 \to^*_{\mathcal{R}, f_\alpha} T'$ is obtained by first applying the path ν on the subtree of T_1 rooted at 1 and then applying the path μ on the resulting tree (in the second step, we keep the subtree of T_1 rooted at the node 0 unchanged). □

We now prove the direction (A1)⇒(A2) of the above claim. The proof is by induction on the length k of the path $\pi : T_0 \to^*_{\mathcal{R}, f} T$. The base case is when $k = 0$, in which case we set $q'(x) := 0$ for all $x \in \mathcal{V}$ and (A2) is satisfied since $I_{\lambda_0(\epsilon), f} = I_{X, f, q'}$.

We proceed to the induction case, i.e., when $k > 0$. We may assume that $\lambda(\epsilon)$ is a *strict* superset of $\lambda_0(\epsilon)$; otherwise, we may replace π with an empty path, which reduces us to the base case. Therefore, some rewrite rule $(g, \texttt{AddClass}(B))$ with $B \cap \lambda_0(\epsilon) \neq \emptyset$ is applied at ϵ at some point in π. We have two cases depending on the first application of such a rewrite rule $\sigma = (g, \texttt{AddClass}(B))$ at ϵ in π:

(Case 1) We have $g = A$ or $g = \langle\uparrow\rangle A$ for some $A \subseteq \mathcal{K}$. Since $\lambda_0(\epsilon) = \lambda'(\epsilon)$, the rule σ can already be applied at ϵ in T_0 yielding a tree $T_1 = (\{\epsilon\}, \lambda_1)$ with $\lambda_1(\epsilon) = \lambda_0(\epsilon) \cup B$. Owing to Lemma 2, we may apply the same sequence of rewrite rules witnessing π but with T_1 as an initial configuration. Furthermore, we may remove any application of σ at ϵ in this sequence (since B is always a subset of the label of ϵ), which gives a path $\pi' : T_1 \to_{\mathcal{R}, f} T$ of length $\leq k-1$. Altogether, we obtain another path $T_0 \to_{\sigma, f} T_1 \to^*_{\mathcal{R}, f} T$ of length $\leq k$. In addition, we have $I_{\lambda_0(\epsilon), f} \to_{\mathcal{P}} I_{\lambda_1(\epsilon), f}$ by applying the rule $(1, \varphi) \in \Delta$ generated from σ in the construction of $\mathcal{P}$. Now we may apply the induction hypothesis to the path $\pi' : T_1 \to^*_{\mathcal{R}, f} T$ (of length $< k$) and obtain $q' : \mathcal{V} \to \{0,1\}$ such that $I_{\lambda_1(\epsilon), f} \to^*_{\mathcal{P}} I_{\lambda(\epsilon), f, q'}$, which gives us a path from $I_{\lambda_0(\epsilon), f}$

to $I_{\lambda(\epsilon),f,q'}$, as desired.

(Case 2) We have $g = \langle\downarrow\rangle A$ for some $A \subseteq \mathcal{K}$. In this case, the first rule applied in π must be of the form $\gamma = (C, \texttt{AddChild}(Y))$ for some $C, Y \subseteq \mathcal{K}$. That is, we have $\pi : T_0 \to_{\gamma,f} T_2 \to^*_{\mathcal{R},f} T_3 \to_{\sigma,f} T_4 \to^*_{\mathcal{R},f} T$ for some trees $T_i = (D_i, \lambda_i)$ ($i \in \{2,3,4\}$) such that: (1) $\lambda_0(\epsilon) = \lambda_i(\epsilon)$ for all $i \in \{2,3\}$, (2) $D_2 = \{\epsilon, 0\}$, (3) At T_3, the rule σ is applied at ϵ, and so $D_4 = D_3$ and $\lambda_4(v) = \lambda_3(v)$ for $v \neq \epsilon$ and $\lambda_4(\epsilon) = \lambda_3(\epsilon) \cup B \supset \lambda_3(\epsilon)$. So, this means that some child $i \in \mathbb{N}$ of the root node in T_3 satisfies $\lambda_3(i) \supseteq A$, which enables σ to be executed at the root node. Now recall that simple guards allow a node to inspect only its immediate neighbours (i.e. parent or children). For this reason, we may assume without a loss of generality that $i = 0$, i.e., that it was the child of ϵ that was spawned by the first transition γ. In fact, we may go one step further to assume that 0 is the only child of the root node in the subpath $T_2 \to^*_{\mathcal{R},f} T_3$! This is because that without modifying the label of ϵ, simple guards do not allow the node 0 to "detect" the presence of the other children.

To finish off the proof, we will apply induction hypotheses twice on two different paths. Firstly, we apply the induction hypothesis on the shorter path $(T_2)_{|0} \to^*_{\mathcal{R},f_\epsilon} (T_3)_{|0}$, where f_ϵ is an assumption function that captures the node label $\lambda_3(\epsilon)$, i.e., $f_\epsilon(\texttt{root}) = 0$ and $f_\epsilon(c) = 1$ iff $c \in \lambda_3(\epsilon)$; note that $\lambda_0(\epsilon) = \lambda_2(\epsilon) = \lambda_3(\epsilon)$. [Recall that $(T_2)_{|0}$ means the subtree of T_2 rooted at the node 0.] This gives us the path $I_{\lambda_2(0),f_\epsilon} \to^*_{\mathcal{P}} I_{\lambda_3(0),f_\epsilon,q''}$ for some $q'' : \mathcal{V} \to \{0,1\}$. Therefore, we have the path $\nu_1 : I_{\lambda_0(\epsilon),f} \to_{\mathcal{P}} (\bar{0}, \mu_\epsilon\mu_0) \to^*_{\mathcal{P}} (q'', \mu_\epsilon\mu'_0) \to_{\mathcal{P}} (q', \mu_\epsilon) \to (q', \mu'_\epsilon)$, where

- $\mu_\epsilon(r) = 1$ iff $I_{\lambda_0(\epsilon),f}(r) = 1$ for each $r \in \mathcal{W} = \{y_c, z_c : c \in \mathcal{K}\} \cup \{\texttt{root}\}$.
- $\mu_0(\texttt{root}) = 0$; $\mu_0(z_c) := \mu_\epsilon(y_c)$; and $\mu_0(y_c) = 1$ iff $c \in B$, for each $c \in \mathcal{K}$.
- $\mu'_0(r) = I_{\lambda_3(0),f_\epsilon,q''}(r)$ for each $r \in \mathcal{W}$.
- $q'(\texttt{pop}) = 1$ and $q'(x_c) = \mu'_0(y_c)$ for each $c \in \mathcal{K}$.
- $\mu'_\epsilon(r) = 1$ iff $\mu_\epsilon(r) = 1$ or $r \in B$.

Note that (q', μ'_ϵ) is the same as $I_{\lambda_4(\epsilon),f,q'}$. We are now going to apply the induction hypothesis the second time. To this end, we let $T'_4 = (\{\epsilon\}, \lambda'_4)$ be the single-node tree defined by restricting T_4 to the root, i.e., $\lambda'_4(\epsilon) = \lambda_4(\epsilon) \supset \lambda_0(\epsilon)$. Since $T_0 \preceq T_4$, by the proof of Lemma 2, we have $T'_4 \to^*_{\mathcal{R},f} T$ by following the actions in the path π, except for the action σ. This gives us a path π' whose length is $|\pi| - 1$. So, by induction, we have a path $\nu_2 : I_{\lambda_4(\epsilon),f} \to^*_{\mathcal{P}} I_{\lambda(\epsilon),f,p}$ for some $p : \mathcal{V} \to \{0,1\}$. In order to connect ν_1 and ν_2, we need to use the following lemma, which completes the proof of Lemma 7.

Lemma 9. *Suppose $p, q : \mathcal{V} \to \{0,1\}$ such that $p(x) \leq q(x)$ for each $x \in \mathcal{V}$. Then, for all stack content $v = \alpha_1 \cdots \alpha_m$ where each $\alpha_i : \mathcal{W} \to \{0,1\}$, if $(p, v) \to_\sigma (p', w)$ with $\sigma \in \Delta$, then $(q, v) \to_\sigma (q', w)$ for some $q' : \mathcal{V} \to \{0,1\}$ such that $p'(x) \leq q'(x)$ for each $x \in \mathcal{V}$.*

Proof. Easy to verify by checking each $\mathcal{P}$-rule. □

K A polynomial-time fixed point algorithm for bounded height

We now provide a simple fixpoint algorithm for the k-redundancy problem that runs in time $n^{O(k)}$. We assume that we have simplified our rewrite systems using the simplification given in Section 5. The input to the algorithm is a simple rewrite system $\mathcal{R}$ over $\textsc{Tree}(\Sigma)$ with $\Sigma = 2^{\mathcal{K}}$,

a guard database $S \subseteq \mathcal{K}$, and an initial Σ-labeled tree $T_0 = (D_0, \lambda_0) \in \textsc{Tree}(\Sigma)$. The number $k \in \mathbb{N}$ is fixed (i.e. not part of the input). Our algorithm will compute a tree $T_F = (D_F, \lambda_F)$ such that $T_F \in \mathit{post}^*_{\mathcal{R}}(T_0)$ and, for *all* trees $T \in \mathit{post}^*_{\mathcal{R}}(T_0)$, we have $T \preceq T_F$. By Lemma 1, if a guard $g \in S$ is matched in some $T \in \mathit{post}^*_{\mathcal{R}}(T_0)$, then g is matched in T_F. The converse also holds since $T_F \in \mathit{post}^*_{\mathcal{R}}(T_0)$. Furthermore, we shall see that T_F is of height at most k, each of whose nodes has at most $|T_0| + |\mathcal{R}|$ children (and so of size at most $O(|T_0| + |\mathcal{R}|)^{O(k)}$), and is computed in time $(|T_0| + |\mathcal{R}|)^{O(k)}$. Computing the rules in S that are matched in T_F can be easily done in time linear in $|S| \times |T_F|$.

The algorithm for computing T_F works as follows.

(1) Set $T = (D, \lambda) := T_0$;

(2) Set $W := \emptyset$;

(3) Add each pair (v, σ) to W, where $v \in D$ and $\sigma \in \mathcal{R}$ is a rule that is applicable at v, i.e. $v, T \models g$ where $\sigma = (g, \chi)$ for some χ; all pairs in W are initially "unmarked";

(4) Repeat the following for each unmarked pair $(v, \sigma) \in W$ with $\sigma = (g, \chi)$:

- **(a)** if $\chi = \texttt{AddClass}(X)$, then $\lambda(v) := \lambda(v) \cup X$;
- **(b)** if $\chi = \texttt{AddChild}(X)$, $|v| \leq k - 1$, and $X \not\subseteq \lambda(v.i)$ for *each* $v.i \in D$ child of v, then set $D := D \cup \{v.(i+1)\}$ where $i := \max\{j \in \mathbb{N} : v.j \in D\} + 1$ (by convention, $\max \emptyset = 0$), and $\lambda(v.i) := X$;
- **(c)** Mark (v, σ);
- **(d)** For each $(w, \gamma) \in (D \times \mathcal{R}) \setminus W$ that is applicable, add (w, γ) to W unmarked;

(5) Output $T_F := T$;

Proof of Correctness. We will show that $T \preceq T_F$ for each $T \in \mathit{post}^*_{\mathcal{R}}(T_0)$. The proof is by induction on the length m of path from T_0 to T. We write $T_F = (D_F, \lambda_F)$. The base case is when $m = 0$, in which case $T = T_0 = (D_0, \lambda_0)$. Since our rewrite rules only add classes to nodes and add new nodes, it is immediate that $D_0 \subseteq D_F$ and that $\lambda_0(v) \subseteq \lambda_F(v)$ for each $v \in D_0$. That is, $T_0 \preceq T_F$. We now proceed to the inductive step and assume that $T = (D, \lambda) \preceq T_F$ with a witnessing embedding $f : D \to D_F$. Let $v \in D$ and $\sigma = (g, \chi)$ be a $\mathcal{R}$-rule that is applicable on the node v of T. By Lemma 1, it follows that $f(v), T_F \models g$. Let us suppose that after applying σ on v, we obtain $T' := (D', \lambda')$. We have two cases now:

1. $\chi = \texttt{AddClass}(X)$. In this case, $D' = D$, $\lambda'(u) = \lambda(u)$ whenever $u \neq v$, and $\lambda'(v) = \lambda(v) \cup X$. Since $T \preceq T_F$, it follows that $\lambda(v) \subseteq \lambda_F(f(v))$. Since $f(v), T_F \models g$, it follows that $(f(v), (g, \texttt{AddClass}(X)))$ is marked in W (by the end of the computation of T_F). This means that $X \subseteq \lambda_F(f(v))$. Altogether, $T' \preceq T_F$.

2. $\chi = \texttt{AddChild}(X)$ and $|v| \leq k - 1$. In this case, $D' = D \cup \{v.i\}$ for some $i \in \mathbb{N}$ with $v.i \notin D$, $\lambda'(u) = \lambda(u)$ for each $u \in D$, and $\lambda'(v.i) = X$. Since $T \preceq T_F$, the properties (H1) and (H3) of embeddings imply that v and $f(v)$ have the same levels in T and T_F respectively. Also, since $v, T_F \models g$, it follows that $(f(v), (g, \texttt{AddChild}(X)))$ is marked in W (by the end of the computation of T_F). This means that there exists a child $f(v).j$ in D_F with $X \subseteq \lambda_F(f(v).j)$. Hence, the extension of f with $f(v.i) := f(v).j$ is a witness for $T' \preceq T_F$, as desired.

Running time analysis. We first analyse the time complexity of each individual step in the above algorithm. Checking whether a simple guard g is satisfied at a node v can be done by inspecting the node labels of v and its neighbours (i.e. parent and children), which can be performed in time $O(|\mathcal{K}| \times (\text{number of neighbours of } v))$. So, since each node in T_0 has most $|T_0|$ children, then step (3) can be achieved in time $O(|\mathcal{K}| \times |T_0|^2)$. Similarly, modifying a node label can be done in time $O(|\mathcal{K}|)$, which is the running time of Step (4a). Since no rewrite rule in $\mathcal{R}$ can be applied twice at a node, each node in T_F has at most $m := |T_0| + |\mathcal{R}|$ children, i.e., either that child was already in T_0 or that it was generated by a rewrite rule in $\mathcal{R}$. Therefore, Step (4b) takes time $O(|\mathcal{K}| \times m)$. Since Step (4d) needs to check only (w, γ) where w is a neighbor (i.e. children or parent) of v, then it can be done in time $O(|\mathcal{K}| \times m)$. Since the tree T_F has height at most k and each node has at most m children, the tree T_F has at most $O(m^k)$ nodes. So, the entire Step (4) runs in time $O(|\mathcal{K}| \times |T_0|^{k+1} \times |\mathcal{R}|^{k+1})$, which is also a time bound for the running time of the entire algorithm.

L W[1]-hardness for k-redundancy problem

Before proving our W[1]-hardness, we first briefly review some standard concepts from parameterised complexity theory [18]. A *parameterized problem* $\mathcal{C}$ is a computational problem with input of the form $\langle v; k\rangle$, where $v \in \{0,1\}^*$ and $k \in \mathbb{N}$ (represented in unary) is called the *parameter*. An *fpt-algorithm* for $\mathcal{C}$ runs in time $O(f(k).n^c)$ for some constant c, and some computable function $f : \mathbb{N} \to \mathbb{N}$ (both independent of input). Here, n is the size of the input. So, an algorithm that runs in time $O(n^k)$ is not an fpt-algorithm. The problem $\mathcal{C}$ is said to be *fixed-parameter tractable* if it can be solved by an fpt-algorithm. There are classes of problems in parameterized complexity that are widely believed not to be solvable by fpt-algorithms, e.g., W[1] (and others in the W-hierarchy). To show that a problem $\mathcal{C}$ is W[1]-hard, it suffices to give an fpt-reduction to $\mathcal{C}$ from the *short acceptance problem of nondeterministic Turing machines (NTM)*: given a tuple $\langle(\mathcal{M}, w); k\rangle$, where $\mathcal{M}$ is an NTM, $w \in \{0,1\}^*$ is an input word, and k is a positive integer represented in unary, decide whether $\mathcal{M}$ accepts w in k steps. Among others, standard reductions in complexity theory that satisfy the following two conditions are fpt-reductions: (1) fpt-algorithms, (2) if k is the input parameter, then the output parameter is $O(k)$.

We now show that k-redundancy is W[1]-hard. To this end, we reduce from the short acceptance problem of nondeterministic Turing machines. The input to the problem is $\langle \mathcal{M}, w; k\rangle$, where $\mathcal{M}$ is an NTM, w is an input word for $\mathcal{M}$, and $k \in \mathbb{N}$ given in unary. Suppose that $\mathcal{M} = (\Gamma, \Sigma, \sqcup, \mathcal{Q}, \Delta, q_0, q_{acc})$, where $\Gamma = \{0,1\}$ is the input alphabet, $\Sigma = \{0,1,\sqcup\}$ is the tape alphabet, $\sqcup$ is the blank symbol, $\mathcal{Q}$ is the set of control states, $\Delta \subseteq (\mathcal{Q} \times \Sigma) \times (\mathcal{Q} \times \Sigma \times \{L, R\})$ is the transition relation (L/R signify go to left/right), $q_0 \in \mathcal{Q}$ is the initial state, and q_{acc} is the accepting state. Each configuration of $\mathcal{M}$ on w of length k can be represented as a word

$$(0, a_0, c_0)(1, a_1, c_1) \cdots (k, a_k, c_k)$$

where $a_i \in \Sigma$ for each $i \in [0, k]$, and for some $j \in [0, k]$ we have $c_j \in \mathcal{Q}$ and for all $r \neq j$ it is the case that $c_j = \star$. Note that this is a word (of a certain kind) over $\Omega := \Omega_k := [0, k] \times \Sigma \times (\mathcal{Q} \cup \{\star\})$. In the sequel, we denote by $\texttt{CONF}_k$ the set of all configurations of $\mathcal{M}$ of length k. Suppose that $w = a_1 \cdots a_n$. If $n < k$, we append some blank symbols at the end of w and assume that $|w| \geq k$. The initial configuration will be $I_0(w) := (0, \sqcup, q_0)(1, a_1, \star) \cdots (k, a_k, \star)$. [Notice that if $n > k$, then some part of the inputs will be irrelevant.] In the following, we will append the symbol $(-1, \sqcup, \star)$ (resp. $(k+1, \sqcup, \star)$) before the start (resp. after the end) of $\mathcal{M}$ configurations to aid our definitions. Therefore, define $\Omega_e := \Omega_{k,e} := [-1, k+1] \times \Sigma \times (\mathcal{Q} \cup \{\star\})$.

We now construct a tree-rewrite system $\mathcal{P}$ to simulate $\mathcal{M}$. First off, for each Δ-rule σ, define a relation $R_\sigma \subseteq \Omega_e^4$ such that $(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{w}) \in R$ describes the update by rule σ of cell $\vec{v}_2$ to $\vec{w}$ given the neighbouring cells $\vec{v}_1$ and $\vec{v}_3$. That is, $(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{w}) \in R$ iff a configuration $C_2 = \vec{u}_{-1} \cdots \vec{u}_{k+1}$ of $\mathcal{M}$, where $\vec{u}_{i+1} = \vec{w}$ for some i, can be reached in one step using σ from another configuration $C_1 = \vec{u}'_{-1} \cdots \vec{u}'_{k+1}$, where $\vec{u}'_i \vec{u}'_{i+1} \vec{u}'_{i+2} = \vec{v}_1 \vec{v}_2 \vec{v}_3$. Let $R = \bigcup_{\sigma \in \Delta} R_\sigma$. In this case, we say that rule σ is a witness for the tuple $(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{w})$. Notice that, given a rule σ of $\mathcal{P}$ and $\vec{v}_1, \vec{v}_2, \vec{v}_3$, there exists *at most one* $\vec{w}$ such that $(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{w}) \in R$ and is witnessed by σ. In addition, it is easy to see that checking whether $(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{w}) \in R$ can be done in polynomial time (independent of k) since this is a simple local check of $\mathcal{M}$ rules.

The rewrite system $\mathcal{P}$ works over trees of labels $\Xi = \Omega_e \cup \Delta$. The initial tree is a tree with a single node labeled by $I_0(w)$. We add the following rules to $\mathcal{P}$:

1. For each $\sigma \in \Delta$, we add $(\top, \texttt{AddChild}(\sigma))$ to $\mathcal{P}$.
2. For each $(\vec{v}_{i-1}, \vec{v}_i, \vec{v}_{i+1}, \vec{w}) \in R$ witnessed by $\sigma \in \Delta$, for $i \in [0, k]$, we add
$$(\sigma \wedge \langle\uparrow\rangle\{\vec{v}_{i-1}, \vec{v}_i, \vec{v}_{i+1}\}, \texttt{AddClass}(\vec{w}))$$
to $\mathcal{P}$.
3. For each $\sigma \in \Delta$, we add $(\sigma, \texttt{AddClass}((-1, \sqcup, \star)))$ and $(\sigma, \texttt{AddClass}((k+1, \sqcup, \star)))$ to $\mathcal{P}$.

Rule 1 first guesses the rule σ to be applied to obtain the next configuration C_2 of the NTM $\mathcal{M}$. This rule is added as a class in the child v of the current node u containing the current configuration C_1. Rule 3 is then applied to obtain the left and right delimiter. After that, configuration C_2 is written down in node v by applying Rule 2 for k times. In particular, the system will look up the rule σ that was guessed before and the content of the parent node u (i.e. C_1). If we let $S = [0, k] \times \Sigma \times \{q_{acc}\}$, then S is not k-redundant iff $\mathcal{M}$ accepts w within k steps. Notice that this is an fpt-preserving reduction. In particular, it runs in time $O(k^c n^d)$ for some constant c and d.

M Proof of PSPACE-hardness in Theorem 2

We reduce to the k-redundancy problem, when k forms part of the input, from membership of a linear bounded Turing machine. Note, since the runs of a linear bounded Turing machine may be of exponential length, the encoding in the previous section no longer works.

Suppose the input TM is $\mathcal{M} = (\Gamma, \Sigma, \sqcup, \mathcal{Q}, \Delta, q_0, q_{acc})$, where $\Gamma = \{a, b\}$ is an input alphabet, $\Sigma = \{a, b, \sqcup\}$ is the tape alphabet, $\sqcup$ is the blank symbol, $\mathcal{Q}$ is the set of control states, $\Delta \subseteq (\mathcal{Q} \times \Sigma) \times (\mathcal{Q} \times \Sigma \times \{L, R\})$ is the transition function (L/R signify go to left/right), $q_0 \in \mathcal{Q}$ is the initial state, and q_{acc} is the accepting state. We are only interested in runs of $\mathcal{M}$ that use $N := dn$ space for some constant d. Each configuration of $\mathcal{M}$ on w can be represented as a word of length $N+1$ in the alphabet $\Omega := \Omega_N$, which was defined in the proof of $\mathsf{W}[1]$-hardness. The initial configuration $I_0(w)$ is defined to be

$$(0, a_0, c_0)(1, a_1, c_1) \cdots (N, a_N, c_N)$$

where $a_0 a_1 \cdots a_N = \sqcup w \sqcup^{(d-1)n}$, $c_0 = q_0$, and $c_1 = \cdots = c_N = \star$. In the following, we will also use extension $\Omega_e := \Omega_{N,e}$ of Ω and the logspace-computability of the 4-ary relation $\Delta \subset \Omega_e^4$ defined in that proof.

We now construct our rewrite system $\mathcal{R}$, which works over the alphabet $\Xi := \{R, 0, 1\} \cup \Omega_e$. Let $M := 2(N + \log|\mathcal{Q}| + \log N)$. Firstly, it suffices to consider runs of $\mathcal{M}$ of at most length

$3^N \times |\mathcal{Q}| \times N < 2^M$ since exceeding this length there must be a repeat of configurations by the pigeonhole principle. Our initial configuration is $T_0 = (D_0, \lambda_0)$, where $D = \{\epsilon\}$ and $\lambda_0(\epsilon) = R$.

We start by adding the following rules to $\mathcal{R}$: $(g, \texttt{AddChild}(0))$ and $(g, \texttt{AddChild}(1))$, where $g = (\langle\uparrow\rangle)^i R$ and $i \in [0, M-1]$. These rules simply build trees of height M, into which the binary tree of height M can be embedded. The label in any path from the root to a leaf node gives us a number i in $[0, 2^M]$ written in binary, which will in turn have a child representing the ith configuration of $\mathcal{M}$ from $I_0(w)$ (which is unique since $\mathcal{M}$ is deterministic).

We now add the rule $(g, \texttt{AddChild}(\{(-1, \sqcup, \star), (N+1, \sqcup, \star)\}))$, where $g = (\langle\uparrow\rangle)^M R$. This rule initialises the content of each configuration (of a given branch).

Notation. *For a guard g, a class c, and direction $d \in \{\uparrow, \downarrow, \uparrow^+, \downarrow^+\}$, we write $c\langle d\rangle g$ to mean $c \wedge \langle d\rangle g$.*

The following set of rules defines the first configuration $I_0(w)$ in the tree. For each $i \in [0, N]$, we add the rule $(g, \texttt{AddClass}((i, a_i, c_i)))$ $g = \langle\uparrow\rangle(0\langle\uparrow\rangle)^{M-1} R$.

We now add a set of rules for defining subsequent configurations in the tree. The guard will inspect a previous configuration of the current configuration. Based on this, it will add an appropriate content of each tape cell. Each of these rules will be of the form $(g, \texttt{AddChild}(\vec{v}))$, where $\vec{v} \in \Omega$. For each $(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{w})$ and $i \in [0, M-1]$, we add a rule $(g, \texttt{AddChild}(\vec{w}))$ where $g := \top\langle\uparrow\rangle\sigma_i\langle\downarrow\rangle\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$ where $\sigma_i = (0\langle\uparrow\rangle)^i 1\langle\uparrow\rangle\top\langle\downarrow\rangle 0(\langle\downarrow\rangle 1)^i$. The guard g selects precisely the nodes containing the jth configuration of $\mathcal{M}$ (there can be at most one jth configuration of $\mathcal{M}$ since $\mathcal{M}$ is deterministic), where j is a number whose binary representation ends with 10^i. The guard traverses the tree upward trying to find the first bit that is turned on (i.e. of form 10^i) and then looks at the complement of this bit pattern (i.e. of form 01^i).

Finally, if we set $S = \{(i, a, q_F) : i \in [0, N], a \in \Sigma\}$, then S is not $(M+1)$-redundant iff $\mathcal{M}$ accepts w. The reduction is easily seen to run in polynomial time. This completes our reduction.

N Proof of EXP-hardness in Theorem 1

We show that the redundancy problem for $\mathcal{R}$ is EXP-hard. To this end, we reduce membership for alternating linear bounded Turing machines, which is EXP-complete. An alternating Turing machine is a tuple $\mathcal{M} = (\Gamma, \Sigma, \sqcup, \mathcal{Q}, \Delta, q_0, q_{acc})$, where $\Gamma = \{0, 1\}$ is the input alphabet, $\Sigma = \{0, 1, \sqcup\}$ is the tape alphabet, $\sqcup$ is the blank symbol, $\mathcal{Q}$ is the set of control states, $\Delta \subseteq (\mathcal{Q} \times \Sigma) \times (\mathcal{Q} \times \Sigma \times \{L, R\})^2 \times \{\wedge, \vee\}$ is the transition relation. Given an $\mathcal{M}$-configuration C, a transition $((q, a), (q_L, a_L, d_L), (q_R, a_R, d_R), s)$ spawns two new $\mathcal{M}$-configurations C_L and C_R such that C_L (resp. C_R) is obtained from C by applying a normal NTM transition $((q, a), (q_L, a_L, d_L))$ (resp. $((q, a), (q_R, a_R, d_R))$). In this way, a run π of $\mathcal{M}$ is a binary tree, whose nodes are labeled by $\mathcal{M}$-configurations and additionally each internal (i.e. non-leaf) node is labeled by a transition from Δ. To determine whether the run π is accepting, we construct a boolean circuit from π as follows: (1) each internal node labeled by a transition $((q, a), (q_L, a_L, d_L), (q_R, a_R, d_R), s)$ is replaced by a boolean operator s, and (2) assign 1 (resp. 0) to a leaf node if the configuration is (resp. is not) in the state q_{acc}. The run π is accepting iff the output of this circuit is 1.

We are only interested in tape configurations of size $d.n$ for a constant integer d, where n is the length of the input word w. Therefore, each configuration of $\mathcal{M}$ on input w is a word in $\texttt{CONF}_N$ defined in the proof of W[1]-hardness. As before, the initial configuration $I_0(w)$ is

$$(0, a_0, c_0)(1, a_1, c_1) \cdots (N, a_N, c_N)$$

where $a_0 a_1 \cdots a_N = \sqcup w \sqcup^{(d-1)n}$, $c_0 = q_0$, and $c_1 = \cdots = c_N = \star$. As before, we will also use extension $\Omega_e := \Omega_{N,e}$ of Ω and put a delimiter at the left/right end of each configuration. For each $\sigma = ((q,a),(q_L,a_L,d_L),(q_R,a_R,d_R),s) \in \Delta$ and $t \in \{L,R\}$, we define a relation $R_{\sigma,t} \subseteq \Omega_e^4$ as the log-computable relation $R_{\sigma'}$ defined in the previous $\mathsf{W}[1]$-hardness proof, where σ' is the NTM transition rule $((q,a),(q_t,a_t,d_t))$.

Our rewrite system $\mathcal{R}$ will work over the set $\mathcal{K} := \{\texttt{root}, \texttt{success}, L, R\} \cup \Omega_e \cup \Delta$ of classes. The initial configuration is $T_0 = (D_0, \lambda_0)$ with $D_0 = \{\epsilon\}$ and $\lambda_0(\epsilon) = \{\texttt{root}, (-1, \sqcup, \star), (N+1, \sqcup, \star)\} \cup \{(i, a_i, c_i) : i \in [0,N]\}$. Define $g := \{\texttt{root}, \texttt{success}\}$, which is the guard we want to check for redundancy. Roughly speaking, our rewrite system $\mathcal{R}$ will guess an accepting run π of $\mathcal{M}$ on the input word w. Each accepting configuration (at the leaf) will then be labeled by `success`. This label `success` will be propagated to the root of the tree according to the $\mathcal{M}$-transition that produce a $\mathcal{M}$-configuration in the tree. More precisely, we define $\mathcal{R}$ by adding the following rewrite rules:

1. For each $t \in \{L,R\}$,
$$\sigma = ((q,a),(q_L,a_L,d_L),(q_R,a_R,d_R),s) \in \Delta$$
and $i \in [0,N]$, add the rule
$$((i,a,q), \texttt{AddChild}(\{\sigma, t, (-1,\sqcup,\star), (N+1,\sqcup,\star)\})$$
to $\mathcal{R}$.

2. For each $\sigma \in \Delta$, $t \in \{L,R\}$, and $(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{w}) \in R_{\sigma,t}$, add the rule
$$(\{t,\sigma\}\langle\uparrow\rangle\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}, \texttt{AddClass}(\vec{w}))$$
to $\mathcal{R}$.

3. For each $i \in [0,N]$ and $a \in \Gamma$, add the rule
$$(\{(i,a,q_{acc}), \texttt{AddClass}(\texttt{success}))$$
to $\mathcal{R}$.

4. For each
$$\sigma = ((q,a),(q_L,a_L,d_L),(q_R,a_R,d_R),s) \in \Delta$$
if $s = \vee$, then add the rule
$$(\langle\downarrow\rangle\{\sigma, \texttt{success}\}, \texttt{AddClass}(\texttt{success}))\ .$$
If $s = \wedge$, then add the rule
$$(g, \texttt{AddClass}(\texttt{success}))$$
where
$$g = \langle\downarrow\rangle\{\sigma, \texttt{success}, L\} \wedge \langle\downarrow\rangle\{\sigma, \texttt{success}, R\}$$

The first rewrite rule spawns a child labeled by an $\mathcal{M}$-transition σ and a symbol $t \in \{L,R\}$ to indicate whether its a left/right child in the $\mathcal{M}$-path to be guessed. The second rewrite rule determines the configuration of the current node (because an $\mathcal{M}$-rule σ has been guessed). The third rewrite rule adds a label `success` to accepting configurations. The fourth rewrite rule propagates the label `success` upwards.

Our translation runs in polynomial-time. Furthermore, it is easy to see that $\mathcal{M}$ accepts w iff g is not redundant with respect to the rewrite system $\mathcal{R}$ and initial tree T_0. This completes our reduction.

O Implementation Optimisations

O.1 Limiting the Number of Rules

For each class $c \in \mathcal{K}$ we construct a symbolic pushdown system that is used to determine whether the class c can be added at a certain node. We improve the performance of the pushdown analysis by first removing all rules of the rewrite system with rules $\mathcal{R}$ that cannot contribute towards the addition of class c at a node. We then perform analysis with the reduced set of rules $\mathcal{R}'$. Since a rule is added for each CSS selector, with intermediate rules being introduced to simplify the guards on the rules, the removal of irrelevant rules to a particular class can give significant gains.

We identify rules which may contribute to the addition of class c by a backwards fixed point algorithm. We begin by including in $\mathcal{R}'$ all rules that either directly add the class c, or add a child to the tree. The reason we add all $\texttt{AddChild}(B)$ rules is because new nodes may lead to new guards being satisfied, even if the label of the new child does not appear in the guard (e.g. $\langle\uparrow\rangle a$ cannot be satisfied without a node labelled with a having a child).

We then search for any rules $\sigma = (g, \texttt{AddClass}(B)) \in \mathcal{R}$ such that there exists $c' \in B$ with c' appearing in the guard of some rule in $\mathcal{R}'$. We add all such σ to $\mathcal{R}'$ and iterate until a fixed point is reached. The full algorithm is given in Algorithm 1

Algorithm 1 Restricting $\mathcal{R}$

Require: A class $c \in \mathcal{K}$ and a set $\mathcal{R}$ of rewrite rules.
Ensure: $\mathcal{R}'$ contains all rules relevant to the addition of c.

$$\mathcal{R}' = \{\ \sigma \in \mathcal{R} \mid \sigma = (g, \texttt{AddClass}(B)) \wedge c \in B\ \} \cup \{\ \sigma \in \mathcal{R} \mid \sigma = (g, \texttt{AddChild}(B))\ \}$$

repeat
  for all $\sigma \in \mathcal{R}'$ and classes c' appearing in σ **do**
    $\mathcal{R}' = \mathcal{R}' \cup \{\ (g, \texttt{AddClass}(B)) \in \mathcal{R} \mid c' \in B\ \}$
  end for
until $\mathcal{R}'$ reaches a fixed point

Proposition 3. *Given a single-node tree T_0, assumption function f and a class $c \in \mathcal{K}$, the rewrite system $\mathcal{R}$ has a positive solution to the class-adding problem $\langle T_0, f, c, \mathcal{R}\rangle$ has a positive answer iff the class-adding problem $\langle T_0, f, c, \mathcal{R}'\rangle$ has a positive answer.*

Proof. The "if" direction is direct since $\mathcal{R}$ contains all rules in $\mathcal{R}'$. The "only-if" direction can be shown by induction over the length of a sequence of rule applications $\sigma_1, \ldots, \sigma_n$ where σ_n adds class c to the tree. Note, we relax the proposition for the induction hypothesis, allowing T_0 to be any arbitrary tree.

In the base case $n = 0$ and the proof is immediate. Hence, assume for the tree T_1 obtained by applying σ_1 to T_0 we have, by induction, a sequence of rule applications $\sigma'_1, \ldots, \sigma'_m$ which is a subsequence of $\sigma_1, \ldots, \sigma_m$, resulting in the addition of class c. We need to show there exists such a sequence beginning at T_0. There are several cases.

When $\sigma_1 = (g, \texttt{AddChild}(B))$ then $\sigma_1 \in \mathcal{R}'$ and the sequence $\sigma_1, \sigma'_1, \ldots, \sigma'_m$ suffices.

When $\sigma_1 = (g, \texttt{AddClass}(B))$ there are three cases. If $c \in B$ then $\sigma_1 \in \mathcal{R}'$ and the sequence σ_1 suffices. If there is some $c' \in B$ used in the matching of some guard g' of a rule in $\sigma'_1, \ldots, \sigma'_m$, then, since c' appears in a rule in $\mathcal{R}'$ we have also $\sigma_1 \in \mathcal{R}'$ and the sequence $\sigma_1, \sigma'_1, \ldots, \sigma'_m$ suffices. Otherwise, no $c' \in B$ is used in the matching of some guard g' in $\sigma'_1, \ldots, \sigma'_m$ and the sequence $\sigma'_1, \ldots, \sigma'_m$ suffices. □

O.2 Limiting the Number of jMoped Calls

By performing global backwards reachability analyses with jMoped we in fact only need to perform one pushdown analysis per class $c \in \mathcal{K}$. Given a set of target configurations, global backwards reachability analysis constructs a representation of the set of configurations from which a target configuration may be reached.

Thus, we begin with the set of target configurations (q, a) where $a = (b_1, \ldots, b_m) \in \{0,1\}^m$ with $b_i = 1$ and $b_i = 1$ indicates the addition of class c to the root node. The global backwards reachability analysis of this target set then gives us a representation of all configurations that can eventually reach one of the targets, that is, add the class c. In particular, we obtain from jMoped a BDD representing all $\langle T_v, f, c, \mathcal{R}\rangle$ that have a positive answer to the class-adding problem. We can then use this BDD to determine whether a given class adding problem for c represents a positive instance without having to use jMoped again.

O.3 Matching Guards Anywhere in the Tree

We can check whether a guard g is not matched anywhere in the tree by checking whether $g \vee \langle\downarrow^+\rangle g$ is redundant. However, during the simplification step this results in the addition of a new class for each guard that has to be propagated up the tree.

We avoid this cost by checking directly whether a guard may be matched anywhere in the tree. After simplification this amounts to checking whether a given class may be matched anywhere in the tree. Hence, after we have saturated T_v, we do a final check to detect whether each class c that does not appear in the saturated tree might still be able to appear in a node created by a sequence of `AddChild`(B) rules. To do this, we perform a similar class adding check as above, except we test reachability of a configuration of the form (q, w) where the top character of the stack w is $a = (b_1, \ldots, b_m) \in \{0,1\}^m$ with $b_i = 1$. That is, the node where c was added does not have to be the root node. We obtain a single BDD per class as above, and check it against all assumption functions f that can represent nodes in the saturated tree.

P HTML5 Translation

We informally describe here our process for extracting rules from HTML5 documents that use jQuery. Note, this translation is for experimental purposes only and does not claim to be a rigorous, systematic, or sound approach. It is merely to demonstrate the potential for extracting interesting rewrite rules from real HTML5 and to obtain some representative benchmarks for our implementation. A robust dataflow analysis could in principle be built, and remains an interesting avenue of future work.

We begin by describing how to translate complete jQuery calls that do not depend on their surrounding code. E.g.

```
$('.a').next().append('<div class="b"/>');
```

does not depend on any other code in the document, whereas

```
$(x).next().append('<div class="b"/>');
```

depends on the value of the `x` variable. We will describe in a later section how we track variables, although we will make mention in when translating jQuery calls section when a particular variable receives a given value. First we describe the translation of CSS selectors and jQuery calls.

P.1 CSS Selectors

We translate the following subset of CSS selectors.

- Each class, element type, or ID is directly translated to a class in our model with the same name.
- Conjunctions of guards are translated to sets of classes. E.g. `div.selected.item` becomes $\{div, selected, item\}$.
- Descendent relations are encoded using $\langle\uparrow^+\rangle$. That is `x y`, where `x` and `y` are selectors with translations into guards g and g' respectively becomes $\langle\uparrow^+\rangle g \wedge g'$.
- Child relations are encoded using $\langle\uparrow\rangle$. That is `x > y`, where `x` and `y` are selectors with translations into guards g and g' respectively becomes $\langle\uparrow\rangle g \wedge g'$.

P.2 jQuery Calls

Given a jQuery call of the form

```
$(s).f(x)....g(y).h(z)
```

where s is a CSS selector string and `f(x)....g(y).h(z)` is a sequence of jQuery calls, we can extract new rules as follows. We recursively translate the sequence of calls `$(s).f(x)....g(y)` where `$(s)` is the base case, and obtain from the recursion a guard reflecting the nodes matched by the calls. Then we generate a rule depending on the effect of `h(z)`.

Note that the call to `$(s)` returns a jQuery object which contains within it a stack of sets of nodes matched. When `s` is just a CSS selector this stack will contain a single element that is the set of nodes matched by `s`. Each function in the sequence `f(x)....g(y).h(z)` receives as its `this` object the jQuery object and obtains a new set of nodes derived from the nodes on the top of the stack. This new set is then pushed onto the stack and passed to the next call in the sequence. There is a jQuery function `end()` that simply pops the top element from the stack and passes the remaining stack to the next call. We model this stack as a stack of guards during our translation.

We start with the base case `$(s)`. In this case we translate the CSS selector `s` into a guard g and return the single-element stack g. Note: this assumes `s` is a string constant. Otherwise we set g to be true (i.e. matches any node).

In the recursive case we have a call `l.f(x)` where `l` is a sequence of jQuery calls and `f(x)` is a jQuery call. We obtain a stack σ by recursive translation of `l` and then do a case split on `f`. In each case we obtain a new stack ρ which is returned by the recursive call. In some cases we also add new rules to our translated model. We describe the currently supported functions below. In all cases, let $\sigma = \sigma' g$. Note, our implementation directly supports $\langle\downarrow^*\rangle\varphi$ which is equivalent to $\langle\downarrow^+\rangle\varphi \vee \varphi$, and similarly for $\langle\uparrow^*\rangle\varphi$.

- `animate(...)`: we return $\rho = \sigma$.
- `addBack()`: let $\sigma = \sigma'' g' g$, we set $\rho = \sigma''(g \vee g')$.
- `addClass(`c`)`: set $\rho = \sigma$ and add the rule $(g, \texttt{AddClass}(\{c\}))$ when c is a string constant (otherwise add all classes).

- `ajax(...)`: we attempt to resolve all functions f in the argument to the call to constant functions (usually this coded directly in place by the programmer). If we are able to do this, we translate f with the `this` variable set to σ. We return $\rho = \sigma$. If we cannot resolve f we ignore it (the body of f will be translated elsewhere with `this` matching any node).

- `append(`s`)`: we attempt to obtain a constant string from s. If s is a concatenation of strings and variables, we attempt to resolve all variables to strings. If we fail, we make the simplifying assumption that variables do not contain HTML code (this can be controlled via the command line) and omit them from the concatenation. If we are able to obtain a constant HTML string in this way we use a sequence of new rules and classes to build that tree at the nodes selected by g. E.g. $(g, \texttt{AddChild}(\{x, a\}))$, $(\{x\}, \texttt{AddChild}(\{b\}))$, $(\{x\}, \texttt{AddChild}(\{c\}))$ where x is a fresh class adds a sub-tree containing a node with class "a" with two children with classes "b" and "c" respectively. In the case where we cannot find a constant string (e.g. if we assume variables may contain HTML strings), we simply add rules to construct all possible sub-trees.

- `bind(..., f)`: we attempt to resolve f to a constant function (usually this is provided directly by the programmer). If we are able to do this, we translate f with the `this` variable set to σ. We return $\rho = \sigma$. If we cannot resolve f we ignore it (the body of f will be translated elsewhere with `this` matching any node)

- `blur(f)`: we treat this like a call to `bind(..., f)`.

- `children(`s`)`: we first obtain from the CSS selector s a guard g' (or true if s is not constant), then we set $\rho = \sigma(\langle\uparrow\rangle g \wedge g')$.

- `click(`f`)`: same as `blur()`.

- `closest(`s`)`: we first obtain from the CSS selector s a guard g' (or true if s is not constant), then we set $\rho = \sigma(\langle\downarrow^*\rangle g \wedge g')$.

- `data(...)`: we return $\rho = \sigma$.

- `each(`f`)`: we attempt to resolve f to a constant, as with `click`. The function f may take 0 or 2 arguments (the second being the matched element). In both cases we translate f with `this` set to σ. If it has arguments, then the second argument is also set to σ. We return $\rho = \sigma$.

- `eq(...)`: we return $\rho = \sigma g$.

- `fadeIn(...)`: we return $\rho = \sigma$.

- `fadeOut(...)`: we return $\rho = \sigma$.

- `focus(f)`: same as `blur()`.

- `end()`: we return $\rho = \sigma'$.

- `find(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant), then we set $\rho = \sigma(\langle\uparrow^+\rangle g \wedge g')$.

- `filter(...)`: we simply return $\rho = \sigma g$.

- `first(...)`: we simply return $\rho = \sigma g$.

- `has(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant), then we set $\rho = \sigma(\langle\downarrow^+\rangle g' \wedge g)$.
- `hover(`f_1`,`f_2`)`: we attempt to resolve f_1 and f_2 to constant functions (else we ignore them to be translated elsewhere with less precise guards). Both f_1 and f_2 take a single argument, which is the element being hovered over. We translate both functions with the argument and the `this` variable set to σ.
- `html(`s`)`: this is handled like `append`.
- `next(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant or not supplied), then we set $\rho = \sigma(\langle\uparrow\rangle\langle\downarrow\rangle g \wedge g')$.
- `nextAll(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant or not supplied), then we set $\rho = \sigma(\langle\uparrow\rangle\langle\downarrow\rangle g \wedge g')$.
- `not(...)`: we ignore the negation and simply return $\rho = \sigma g$.
- `on(`e`,`s`,`f`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant or not supplied), and we attempt to resolve f to a constant function (like `click`). We then translate f with `this` set to $\sigma(\langle\uparrow^+\rangle g \wedge g')$. We return with $\rho = \sigma$.
- `parent(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant or not supplied), then we set $\rho = \sigma(\langle\downarrow\rangle g \wedge g')$.
- `parents(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant or not supplied), then we set $\rho = \sigma(\langle\downarrow^+\rangle g \wedge g')$.
- `prepend(`s`)`: this is handled like `append`.
- `prev(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant or not supplied), then we set $\rho = \sigma(\langle\uparrow\rangle\langle\downarrow\rangle g \wedge g')$.
- `prevAll(`s`)`: we obtain from the CSS selector s a guard g' (or true if s is not constant or not supplied), then we set $\rho = \sigma(\langle\uparrow\rangle\langle\downarrow\rangle g \wedge g')$.
- `remove(`s`)`: this function is ignored and we return $\rho = \sigma$.
- `removeClass(`s`)`: this function is ignored and we return $\rho = \sigma$.
- `resize(f)`: same as `blur()`.
- `show(...)`: we return $\rho = \sigma$.
- `slideUp(...)`: we return $\rho = \sigma$.
- `slideDown(...)`: we return $\rho = \sigma$.
- `stop(...)`: we return $\rho = \sigma$.
- `val(...)`: we return $\rho = \sigma g$.

We also ignore the following functions that do not return a jQuery object or manipulate the DOM tree: `extend()`, `hasClass()`, `height()`, `is()`, `width()`.

P.3 More Complex JavaScript

In reality, it is rare for jQuery statements to be completely independent of the rest of the code. We extend our translation by identifying common jQuery programming patterns to enable the generation of more precise rules. This is, in effect, a kind of ad-hoc dataflow analysis over the syntax tree of the JavaScript.

We remark that current JavaScript static analysers do not track the kind of information needed to build meaningful guards from the jQuery calls. Indeed, implementing a systematic dataflow analysis for propagating jQuery guards remains an interesting avenue of future research.

Often a call will take the form

```
$(x).f(y)
```

for some variable `x`. In the worst case one can assume any value for guards derived from `x` by using the guard $\top$. In other cases it is possible to be more precise.

When translating a call such as

```
$('.a').each(function () {
    $(this).addClass('b');
));
```

for example, is it easy to infer that the selection returned by the `this` variable is all elements with the class "a". Hence, when we are translating constant functions such as these, we track the guards in the `this` variable as described in the previous section. Similarly, if we wrote

```
$('.a').each(function (i, e) {
    $(e).addClass('b');
));
```

we pass the stack of guards from the `$('a')` selector to the `e` variable, and are thus able to interpret `$(e)`.

Note, we are also able to translate `$(`s`, x)` where `x` is `this` or some variable known to hold a jQuery object with stack σg by using the guard

$$\langle\uparrow^+\rangle g \wedge g'$$

where g' is the guard obtained from the selector s. Similarly `$(document)` can be translated to the root node of the HTML.

We can further improve the analysis by tracking variables declared in the JavaScript. To this end, we identify all variables that are assigned outside of a loop or conditional variable (i.e. assigned only once) and remember their value for use in later translations. This captures common cases such as

```
var eles = $('myelements');
...
eles.addClass('b');
eles.append('<p>Some test.</p>');
```

Finally, we also provide some support for user-defined jQuery functions. In a preprocessing step, we look for lines of the form

```
$.fn.f = function (args) { ... };
```

and keep a map from `f` to the function definition. Thus, when we find elsewhere in the code a jQuery call of the form

```
...f(params)...
```

we can translate the function body of `f` with the `this` variable set to the current jQuery stack, any parameters can be passed to the arguments of the function. Note, if we discover a line of `f` which overwrites a parameter variable, we will forget the value passed from the call and use a top value instead.