Automated Reasoning for Web Page Layout

Pavel Panchekha Emina Torlak
University of Washington, USA
{pavpan,emina}@cs.washington.edu

Abstract

Web pages define their appearance using Cascading Style Sheets, a modular language for layout of tree-structured documents. In principle, using CSS is easy: the developer specifies declarative constraints on the layout of an HTML document (such as the positioning of nodes in the HTML tree), and the browser solves the constraints to produce a box-based rendering of that document. In practice, however, the subtleties of CSS semantics make it difficult to develop stylesheets that produce the intended layout across different user preferences and browser settings.

This paper presents the first mechanized formalization of a substantial fragment of the CSS semantics. This formalization is equipped with an efficient reduction to the theory of quantifier-free linear real arithmetic, enabling effective automated reasoning about CSS stylesheets and their behavior. We implement this reduction in Cassius, a solver-aided framework for building semantics-aware tools for CSS. To demonstrate the utility of Cassius, we prototype new tools for automated verification, debugging, and synthesis of CSS code. We show that these tools work on fragments of real-world websites, and that Cassius is a practical first step toward solver-aided programming for the web.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

Keywords Solver-aided tools, cascading style sheets, SMT, synthesis, layout

1. Introduction

Cascading Style Sheets (CSS) is the language for specifying the appearance of web pages. It provides developers with a modular, declarative mechanism for controlling all aspects of web page layout—from font sizes, to colors, to positioning of web page elements. Almost all web pages use CSS (W3Techs 2015), with tens of thousands of developers writing millions of lines of CSS code. But writing high-quality CSS is not easy. Because of the subtleties of CSS semantics, many web pages violate their intended layouts when rendered with particular screen sizes, browsers, or user preferences.

```
<html> <body>
  <p>Our products:</p>
  <main>
    <div>A</div> <div>B</div>
    <div>C</div> <div>D</div>
  </main>
  <p>Buy now!</p>
</body> </html>

-----------------------------

main {background:gray;
  width:100%; float:left}
div {height:200px; width:200px;
  float:left; font-size:144pt; }
```

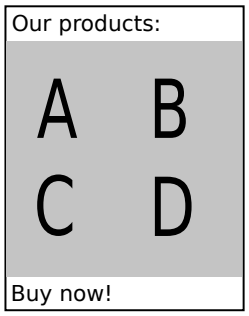

Figure 1: HTML and CSS for a grid of products. An HTML file (above the line on the left) or *document* defines a tree of elements; the CSS file or *stylesheet* below the line has two rules each of which sets several properties. The resulting rendering is shown on the right.

For example, consider the toy product page in Figure 1. It displays a grid of products in a box with a colored background, and the grid expands when the browser window does. The CSS that achieves this layout uses *floating* in two radically different ways. Floating allows two boxes to be horizontally adjacent, as when the products are floated to lay them out in rows. However, the `<main>` container is also floated, even though no boxes are horizontally adjacent to it. Without being made to float, the container would shrink, and the product grid would not have a background color. This is because the container has to be a *flow root* in order to expand to contain its floating children. No dedicated CSS property turns a box into a flow root, but floating the container does so as a side effect. While floating the container works in our case, it would not work if the paragraph following the product grid had a background, or if the container had a border; in both cases, a different fix would be needed. Reasoning through these cases is complex and error-prone, and no tools currently exist to simplify the process.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

OOPSLA '16, October 30-November 04 2016, Amsterdam, Netherlands
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4444-9/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2983990.2984010

Solving a problem like this automatically requires tools—such as verifiers, debuggers, and synthesizers—that can reason about the semantics of web page layout. A web page verifier, for example, could ensure that an off-the-shelf grid layout[1] expands as desired for all screen sizes. A debugger could identify the parts of the stylesheet that are responsible for any problems found by the verifier. Finally, a synthesizer could automatically repair the buggy constraints to enforce the desired properties. Alternatively, if the developer is crafting a web page from scratch, a synthesizer could produce the stylesheet automatically, given the HTML documents to be styled and a mock-up of the desired page layout.

However, *semantic* tools like these do not exist. In fact, the only CSS tools available to developers besides browsers are syntactic linters and compressors. The lack of semantic tools, despite their potential impact and large user base, is best explained by the complexity of the CSS semantics, which makes building such tools both technically challenging and time-consuming.

This paper presents Cassius, the first framework for automated reasoning about web page layout. The framework facilities rapid development of semantic tools for verification, debugging, and synthesis of CSS stylesheets. The key ingredient of Cassius is a faithful and efficient formalization of a substantial fragment of the CSS semantics in the theory of quantifier-free linear real arithmetic (LRA). This formalization enables reasoning about CSS using off-the-shelf Satisfiability Modulo Theories (SMT) solvers.

The Cassius fragment of CSS is rich enough to lay out fragments of many real-world websites, such as Wikipedia, Google, and Amazon. Our formalization focuses on the core aspects of the CSS semantics: cascading stylesheets, generating boxes for HTML elements, and laying out the resulting boxes. Each of these steps is modeled in detail, including computing used values, collapsing margins, and aligning text. In essence, Cassius is a feature-rich, declarative implementation of a browser layout engine, consisting of a set of LRA constraints that relate the inputs to the browser (the CSS stylesheet and the HTML document) to its output (the resulting box-based layout).

Applying Cassius to real web pages involves two key technical innovations. The first is identifying a fragment of CSS that is both expressive and amenable to layout with a *non-deterministic incremental algorithm*. The second is developing a technique, based on *auxiliary uninterpreted functions*, for encoding an arbitrary run of this algorithm in LRA. Together, these two innovations enable Cassius to reduce a given layout problem to a quantifier-free formula that is linear in the size of the problem. In contrast to a naive encoding, which is quadratic in problem size and quickly overwhelms the solver, our encoding can be solved in seconds, even for large inputs.

[1] From a framework such as Bootstrap (Otto and Thornton 2015).

We demonstrate the utility and efficiency of Cassius by using it to prototype three novel semantic tools for CSS. The first tool is a verifier, which checks CSS stylesheets against usability and accessibility properties, such as ensuring that no boxes cover buttons or other control elements. The second tool is a debugger, which highlights the parts of a stylesheet that are responsible for a particular layout outcome, such as the position of a given box on the screen. The third tool is a synthesizer, which generates CSS from the desired layout of several concrete HTML documents. Our synthesizer could be extended to allow designers to work in a visual editor and automatically transfer their designs to the web. By building on top of Cassius, each of these tools was developed in a few days, and they all scale to fragments of real web pages.

In summary, this paper makes the following contributions:

- The first mechanized formalization of a core fragment of the CSS semantics, with an efficient encoding in the theory of linear real arithmetic.
- An implementation of this formalization in Cassius, a framework for building semantic tools that can reason about web page layout.
- Three prototype tools built with Cassius, which demonstrate the utility and efficiency of our formalization.

The rest of the paper is organized as follows. Section 2 presents an overview of Cassius and its applications. Section 3 describes our formalization of the CSS semantics, highlighting the design choices that enable efficient reduction of this semantics to LRA. We show how to perform such a reduction in Section 4. Section 5 demonstrates the conformance of our CSS semantics to browser behavior, its practicality as a subset CSS, and its amenability to efficient automated reasoning. We conclude the paper with a discussion of related work (Section 6) and a summary of contributions (Section 7).

2. Overview and Applications

This section provides an overview of the Cassius framework from the point of view of a tool builder. We describe the interface exposed by the framework and its application to three example tools for web design. Sections 3 and 4 present the details of the framework's semantics and its implementation by reduction to SMT.

2.1 The Cassius Framework

Cassius can be thought of as a declarative browser that exposes an interface similar to that of a standard, imperative browser. Given an HTML document and a CSS stylesheet, an imperative browser produces a *layout* that displays the document contents in a tree of boxes, as specified by the stylesheet. Figure 1 shows an example document, stylesheet, and layout. The interface to Cassius is based on the same concepts: a tree-structured HTML document, a CSS stylesheet that *styles* the nodes in this tree, and a box layout of the tree that respects the given style constraints. Unlike an imperative

browser, however, Cassius can perform the layout computation both forwards and backwards: it can not only compute a layout from a document and a stylesheet, but it can also compute a stylesheet from a document and a desired layout.

To enable reversible layout, Cassius operates on *symbolic* layouts and stylesheets—both may be *sketches* that contain unknown values or *holes* to be filled by the declarative browser. (An imperative browser, in contrast, allows only layouts to have holes.) Figure 2 shows an example input to Cassius, consisting of an HTML document `A`, stylesheet `B`, and symbolic layout `C`. Cassius takes inputs in a simple s-expression syntax, with question marks (`?`) denoting holes. It fills these holes by reducing the layout computation problem to a set of constraints in linear real arithmetic (LRA), solved with Z3 (De Moura and Bjørner 2008). Solving for the holes in Figure 2, for example, is akin to rendering the HTML and CSS from Figure 1.

In general, Cassius takes as input a symbolic stylesheet, a set of document-layout pairs, and a set of LRA *assertions* on the holes (expressing, e.g., usability properties). Given these inputs, it will fill the holes in the stylesheet and layouts so that the resulting concrete stylesheet simultaneously renders each document to its layout, while satisfying the provided assertions. Our declarative browser can thus execute the layout process forwards and backwards for multiple documents that share the same stylesheet.

A declarative browser provides a convenient platform for building a wide variety of semantic tools for web developers. We demonstrate three such tools below and evaluate them on fragments of real web pages in Section 5. All three tools are built by reducing a development task—such as repairing a broken stylesheet—to the problem of completing holes in a symbolic stylesheet or layout. Cassius either completes the holes or returns an explanation for why a completion does not exist, given in terms of CSS constraints. The client tool then presents this output to the user.

```
(document A
  ([html]
   ([body]                        (stylesheet B
    ([p] "Our products:")           ((tag main)
    ([main]                          [width (% 100)]
      ([div] "A")                    [float left])
      ([div] "B")                   ((tag div)
      ([div] "C")                    [width (px 200)]
      ([div] "D"))                   [height (px 200)]
    ([p] "Buy now!"))))              [float left]))

(layout C
  ([ROOT :w 400 :h 600]
   ([BLOCK :w ? :h ? :x ? :y ?]
    ([BLOCK :w ? :h ? :x ? :y ?]
     ([BLOCK :w ? :h ? :x ? :y ?]
      ([LINE :w ? :h ? :x ? :y ?]
       ([TEXT :w 112 :h 19 :x 0 :y 16])))
     ([BLOCK :w 400 :h 400 :x ? :y ?]
      ([BLOCK :w 200 :h 200 :x 0 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 200 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 0 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 200 :y ?] ...))
     ([BLOCK :w ? :h ? :x ? :y ?]
      ([LINE :w ? :h ? :x ? :y ?]
       ([TEXT :w 76 :h 10 :x 0 :y 451])))))))))
```

Figure 2: Figure 1 translated to Cassius input, containing an HTML file, a CSS file, and a rendering (line and text boxes for products are elided for space). All parts of the CSS file or the rendering, except the width and height of text boxes (see Section 3.1), can be replaced by holes (`?`), and Cassius will solve for the values of these holes automatically.

2.2 Verifying Layouts

As we saw in Section 1, using CSS to specify an attractive and usable web page layout can be surprisingly tricky. Even expert-written CSS (e.g., Bootstrap (Otto and Thornton 2015)) can suffer from usability bugs, such as overlapping text or control elements (buttons), figures separated from their captions, or elements rendered off-screen on smaller devices. These bugs may only manifest at a particular screen size, making them difficult to discover. In practice, developers spot-check for stylesheet bugs manually, by using an imperative browser to render their web page at a few sample screen sizes, text sizes, etc. Such manual testing is both tedious and unlikely to achieve good coverage.

Using a declarative browser, we can instead *verify* that a stylesheet satisfies desired properties. Given a document D, a stylesheet C, and a property P, a verifier can prove that P is satisfied when rendering D and C under any possible viewport sizes, font sizes, or other layout parameters unspecified by C. Building on Cassius, we prototyped the first such verification tool for CSS. Figure 3 shows an example stylesheet C (3a) and property P (3b) consumed by our verifier. Given these inputs, the verifier encodes $\neg P$ as an assertion on the holes in the symbolic layout B that corresponds to C and D. The holes in the layout represent parameters that are unspecified by C—in our case, the size of the browser window, or *viewport*. The resulting declarative layout problem is passed to Cassius, which searches for a completion of the holes that violates P. If a completion exists, the verifier converts it to a concrete counterexample layout (Figure 3d) and presents the counterexample to the developer. If not, all possible renderings of C and D are guaranteed to satisfy P.

2.3 Debugging Layouts

To repair a layout bug, such as the product container being too small in Figure 3d, the developer first needs to isolate its cause—a fault in the input stylesheet or document. In this paper, we focus on stylesheet faults.[2] Localizing these faults with an imperative browser involves a manual modify-and-check process, in which individual stylesheet constraints are

[2] We believe, however, that our framework can be extended to support localization of faults in the document structure as well.

```
(stylesheet B
  ((tag main)
   [width (% 100)])
  ((tag div)
   [width (px 200)]
   [height (px 200)]
   [float left]))
```

(a) An incorrect stylesheet: no float property on main.

$$\forall a \in \mathcal{B}, b \in \mathcal{B},$$
$$a.\mathsf{elt.tag} = \mathsf{main}$$
$$\wedge\, b.\mathsf{parent} = a$$
$$\implies \mathsf{contains}(a, b)$$

(b) Checking that all product icons lie within the grid container.

```
(layout C
  ([VIEW
    :w (between 200 1920)]
   ([BLOCK ...]
    ([BLOCK ...]
     ([BLOCK :w ? ...] ...)
     ([BLOCK ...] ...)
     ([BLOCK ...] ...)))))
```

(c)

```
(layout counterexample
  ([VIEW :w 400]
   ([BLOCK ...]
    ([BLOCK ...]
     ([BLOCK ...] ...)
     ([BLOCK :w 400 :h 0
       :x 0 :y 16]
      ...)
     ([BLOCK ...] ...)))))
```

(d)

Figure 3: A prototype CSS verification tool built on top of Cassius. The same HTML document as in Figure 1 is used, but with an incorrect stylesheet (a) that fails to guarantee that the container expands to contain all products (b). Given a choice of the browser window width (c), the verifier finds a counterexample (d) where the property does not hold.

weakened or strengthened until the original bug disappears from the resulting layout, without introducing new bugs.

With a declarative browser, the fault localization process is automated by *unsatisfiable core extraction*. Using Cassius, we built a prototype debugger that takes as input a concrete stylesheet C, document D, layout B, and the layout parameters $\vec{v}$ in B (e.g., the height of the product container in Figure 3d) that violate the desired property $P(B(\vec{v}))$. The debugger converts these inputs into an unsatisfiable layout problem in two steps. First, it translates B into a symbolic layout B' by replacing the value of each $v_i \in \vec{v}$ with a fresh hole h_i. Second, it generates the assertion $P(\vec{h}) \wedge \bigwedge_{h_i \in \vec{h}} h_i = B(v_i)$. By construction, there is no way to complete $\vec{h}$ so that this assertion is satisfied for B', C, and D. Invoking Cassius on such a problem produces an unsatisfiable core—a small subset of the constraints in C, and in the CSS semantics, that is responsible for the violation of $P(\vec{v})$. Our debugger projects this core onto C and pretty-prints it as shown in Figure 4.

2.4 Synthesizing CSS from Examples

Having detected and localized the fault in our toy web page (Figures 3–4), we now turn to the problem of repairing the stylesheet so that the product container expands to contain the product icons. To automate this task, we used Cassius to implement a *CSS sketching* tool that takes as input a stylesheet with holes and a set of document-layout examples,

```
main {background:gray; height; float}
div { float:left; font-size:144pt;
  width:200px; height:200px; }
```

Figure 4: CSS debugger output for the stylesheet, document, and counterexample from Figure 3. The localized fault consists of both *set* properties (the float property of), and *unset* properties (the float and height properties of <main>).

```
main {background:gray;
  height:?; float:?}
div {float:?;
  font-size:144pt;
  width:200px;
  height:200px; }
```

(a) A stylesheet sketch with expression holes.

```
main {background:gray;
  float: left}
div {float:left;
  font-size:144pt;
  width:200px;
  height:200px; }
```

(b) The completed stylesheet found by a synthesizer.

Figure 5: The stylesheet properties identified by a CSS debugger (Figure 4) are replaced with expression holes, which the synthesizer fills. Note that the synthesizer ultimately decides not to set the height property.

and fills the holes so that the resulting stylesheet renders each document to the corresponding layout. A stylesheet sketch (Figure 5a) contains holes which stand for unknown CSS syntax, while a document-layout example consists of a document and its concrete rendering. The synthesizer translates the stylesheet sketch and the document-layout examples into a declarative layout problem using a standard reduction (Solar-Lezama et al. 2006) of expression holes to numeric holes. If Cassius finds values for the numeric holes, the synthesizer lifts them into a valid completion of the stylesheet sketch. In our running example, the synthesizer repairs the buggy stylesheet from Figure 3a to produce the stylesheet shown in Figure 5.

3. CSS Semantics

A key feature of Cassius is a declarative formalization of a substantial fragment of the CSS semantics. The complete formalization is publicly available in machine-readable form [3]. This section presents the core concepts of web page layout, as formalized in Cassius, highlighting design decisions that enable efficient encoding of the layout process in the theory of linear real arithmetic.

Cassius derives its specification of CSS layout from the W3C CSS 2.1 standard (W3C 2007). The standard describes CSS via an abstract algorithm (Figure 6). Given an HTML document and a CSS stylesheet, the algorithm computes a *layout* (6d), which is a set of boxes with known position and size. This computation takes as input the document,

[3] At https://github.com/uwplse/cassius

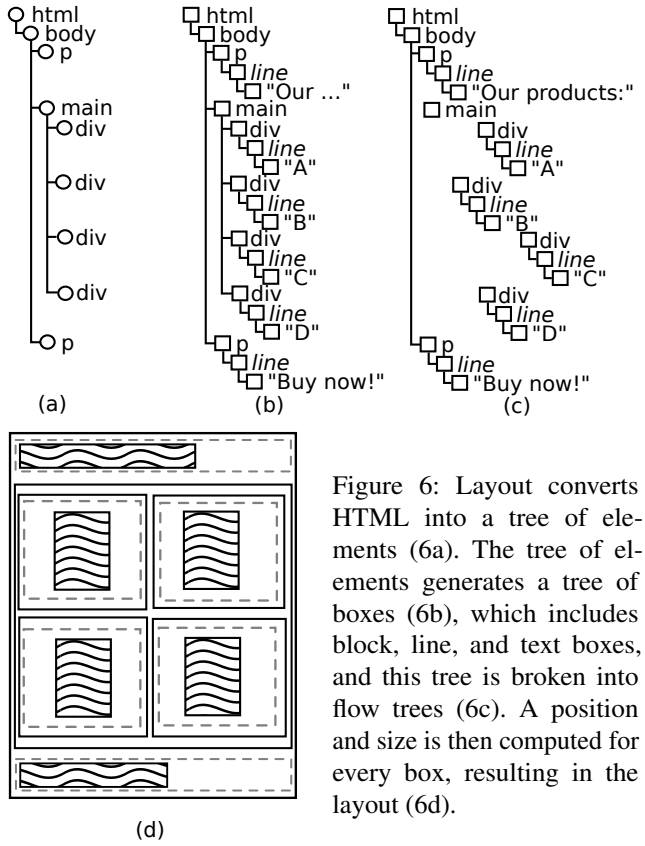

Figure 6: Layout converts HTML into a tree of elements (6a). The tree of elements generates a tree of boxes (6b), which includes block, line, and text boxes, and this tree is broken into flow trees (6c). A position and size is then computed for every box, resulting in the layout (6d).

represented by a tree of elements (6a); constructs a tree of *layout boxes* from the element tree and stylesheet rules (6b); and arranges the resulting boxes into *flow trees* (6c), from which sizes and positions for each box (i.e., the layout) can be computed. Cassius is a specification of this abstract layout algorithm, describing the values computed for each box and the rules by which these values are computed.

The Cassius specification is declarative: it specifies a relation on the HTML elements $\mathcal{E}$, CSS rules $\mathcal{R}$, and the boxes $\mathcal{B}$ that form the final layout. As demonstrated in Section 2, such a declarative formalization provides a versatile platform for building semantic tools: it can be used for verification (by specifying $\mathcal{E}$ and $\mathcal{R}$ and solving for $\mathcal{B}$), synthesis (by specifying $\mathcal{B}$ and $\mathcal{E}$, and sketching $\mathcal{R}$), or debugging (by specifying $\mathcal{E}$, $\mathcal{R}$, and $\mathcal{B}$, and computing unsatisfiable cores). In the rest of this section, we describe the fragment of CSS implemented by Cassius; the representation of $\mathcal{E}$, $\mathcal{R}$, and $\mathcal{B}$; our formalization of the CSS layout algorithm; and the light but crucial restrictions on the use of floating boxes that make automated reasoning tractable.

3.1 Supported Fragment of CSS

Cassius specifies layout for a fragment of CSS level 2.1. The specification covers the cascading rules and the box model, including the details of box generation, block and inline boxes, floating and in-flow boxes, and line boxes. This forms a substantial fragment of CSS, and suffices to lay out fragments of real-world websites. With the fragment of CSS currently specified, additional CSS features could be added in a straightforward manner.

The Cassius formalization focuses on the core layout algorithm. As such, it omits the semantics of CSS rules that do not affect layout (e.g., font and color rules). It also omits the modeling of layout features that are difficult to formalize but easy for a client tool to provide (e.g., font metrics, line breaking, and hyphenation). Finally, the CSS standard describes a rich universe of CSS selectors; Cassius models only tag name, identifier, and universal selectors. Our implementation simply ignores the CSS properties that are not modeled—they can appear in an input to the Cassius declarative browser, but their semantics will not be considered by the underlying SMT solver.

3.2 Representing Elements, Rules, and Boxes

Cassius specifies CSS layout as a relation between a tree of elements $\mathcal{E}$, CSS rules $\mathcal{R}$, and the layout $\mathcal{B}$. The input to Cassius is a representation of these three entities. The elements $\mathcal{E}$ are given by an abstract syntax tree (Figure 2a), where each node is labeled with an HTML *tag* and an optional *identifier*, or is a string of text. The CSS rules $\mathcal{R}$ are given by a list of rule blocks (Figure 2b), each of which contains a *selector* and a collection of property-value pairs. The selector identifies the elements in $\mathcal{E}$ that are styled by a given rule. The selector and properties can be given as *holes*—undefined values that Cassius will solve for. Finally, the layout $\mathcal{B}$ is represented by a set of boxes, arranged in a tree (Figure 2c). Each box is annotated with the box type (root, block, inline, line, or text); the element $e \in \mathcal{E}$ from which it was generated;[4] its position, width, and height; and the width of the border on each side. Any of these fields can be holes except width and height on text boxes.[5]

3.3 CSS Layout

The Cassius specification of CSS layout follows the high-level structure of the abstract layout algorithm described in the CSS standard.

CSS Rules A rule $r \in \mathcal{R}$ is a map from a subset of the CSS properties $\mathcal{P}$ to a value for each property. We write $p \in r$ to denote that the rule r specifies a value for the property p, and we write $r[p]$ for that value. The value $r[p]$ is either the distinguished constant $\mathsf{inherit}$ or it is drawn from a property-specific type. Each rule has an associated selector, $r.\mathsf{selector}$, which can be the universal selector $*$, an HTML tag, or an identifier. Selectors define which elements a rule applies to. Rules are also associated with metadata (such as

[4] Some boxes are not generated by an element, such as line and text boxes or "anonymous" block box. These boxes are annotated as anonymous.

[5] Since Cassius does not model font metrics, the width and height of text boxes must be provided by tools that build upon Cassius, for example by rendering the text and measuring its size.

their position in the stylesheet) that influence cascading, as described below.

Elements Every element has a tag and an optional identifier, which are modeled as opaque symbols. A tag specifies the type of an element, while an identifier gives it a unique name. Both tags and identifiers are used for determining which rules apply to an element: the wildcard selector applies to all elements, and the tag name and identifier selectors apply to those elements that have the given tag name or identifier. Elements are arranged in an ordered tree, giving the usual meaning to the notions of the next, previous, first, and last siblings of an element. Each element also has a *computed style*, which maps every CSS property to a value. We write $e[p]$ for the computed value of the property p for the element e. This value is derived from the rules $\mathcal{R}$ through a process called *cascading*.

Cascading The cascading process determines the computed value $e[p]$ of every element $e \in \mathcal{E}$ and property $p \in \mathcal{P}$. Cassius implements this process declaratively. If no rule in $\mathcal{R}$ both applies to e and specifies a value for p, then $e[p]$ is set to a property-specific default value. Otherwise, $e[p] = r[p]$, where r is the highest scoring rule that sets the property p and whose selector matches e, ranked according to the scoring function from Section 6.4.3 of the CSS standard. This function uses rule metadata to break ties between rules with equally specific selectors. The inherit value for CSS properties is also resolved during cascading.

Boxes The CSS standard defines three types of boxes—block, line, and inline. For modeling convenience, Cassius defines four additional types of boxes: root boxes, which represent browser windows; text boxes, which abstract text rendering; and *opaque boxes*, which abstract the layout of elements (e.g., tables) that are outside of our fragment of the CSS semantics. The root and text boxes are implicit in the CSS standard.

Like elements, boxes are arranged in an ordered tree, giving each box a parent, children, and siblings. Every box also has a width, height, $(x, y) \in \mathbb{R}^2$ position, and an element from which it was generated. These values are computed based on the *margin*, *padding*, and *border* width in each direction, as specified by the *CSS box model*. Cassius models all parts of a box explicitly, as well as the rules for computing box layout, which differ for block boxes, inline and text boxes, and line boxes.

Block Boxes Block boxes can be either *in-flow* or *floating*, and they are generated from block-level elements, such as paragraphs (<p>). In-flow boxes take up all the available horizontal space (in their parent container) and are laid out one after another vertically. Floating boxes are placed to the left or right of other boxes, with text flowing around them.

The CSS standard prescribes a complex set of rules for computing the positions and sizes of block boxes. The computation involves partitioning the box tree into a forest of *flow trees*, computing *used values* of box properties, and collapsing margins of certain boxes. Imperative browsers must carefully sequence this computation. For example, the width of boxes is determined in a top-down pass, while their height is determined in a bottom-up pass, since the height of an element might depend on the size and position of its children.

Because the Cassius semantics is declarative, it dispenses with the need for sequenced computation of box properties. Instead, it simply constrains the final value of each property with a formula. For example, Cassius specifies the x position of an in-flow block box to be the sum of the parent's x position, left padding and border, plus the box's left margin:

$$\forall b \in \mathcal{B}, \mathsf{block\text{-}box}(b) \wedge \mathsf{in\text{-}flow}(b) \implies b.\mathsf{x} = b.\mathsf{parent.left\text{-}content\text{-}edge} + b.\mathsf{margin\text{-}left}$$

Our machine-readable specification of block layout distills 36 pages of the CSS standard into just 790 lines of code.

Line Boxes Line boxes are generated when a block box contains inline or text boxes. Each line box represents a line of text, and all of its siblings are also line boxes. Usually, a line box spans from the left to right edge of its parent's content area. However, line boxes shrink to avoid floating boxes. For example, the following constraints forces line boxes to avoid block boxes that float to the right:

$$\begin{aligned}\forall b \in \mathcal{B}, f \in \mathcal{B}, \mathsf{line\text{-}box}(b) \wedge f.\mathsf{element}[\mathsf{float}] = \mathsf{right} \wedge \\ \mathsf{is\text{-}preceding\text{-}floating\text{-}box}(f, b) \implies \\ \textbf{if } b.\mathsf{top\text{-}margin\text{-}edge} < f.\mathsf{bottom\text{-}margin\text{-}edge} \\ \textbf{then } b.\mathsf{right\text{-}margin\text{-}edge} = f.\mathsf{left\text{-}margin\text{-}edge} \\ \textbf{else } b.\mathsf{right\text{-}margin\text{-}edge} = p.\mathsf{right\text{-}content\text{-}edge}\end{aligned}$$

Line boxes also align their children to the left, right, or center, depending on their parent's text-align property.

Browsers generate line boxes with a *line breaking* algorithm, which breaks text into lines to achieve a visually pleasing result. Different browsers use different line breaking algorithms, and the CSS 2.1 standard does not specify any constraints on this algorithm. As a result, Cassius does not constrain line breaking in any way, instead relying on the input to contain pre-broken lines.[6] This modeling choice does not affect either synthesis or debugging, since the full layout (including the desired line breaks) is available in these tasks. It may, however, cause a verification tool to produce false positives—layouts that violate a desired property under our declarative semantics but not on any imperative browsers, which constrain line breaking. We have found such false positives to be rare in practice (see Section 5).

[6] Since line breaking depends on the fonts used on the page, the language the page is written in, and dictionaries of language-specific layout rules, modeling line breaking in Cassius would be difficult and would likely cause severe performance degradation.

Inline and Text Boxes Inline and text boxes are always descendants of line boxes. They are laid out left to right, each lying to the right of the previous box. Cassius specifies this horizontal layout as follows:

$$\forall b \in \mathcal{B}, \text{text-box}(b) \wedge \text{is-box}(b.\text{previous}) \implies b.\text{left-border-edge} = b.\text{previous.right-border-edge} \quad (1)$$

Cassius does not model the height and width computation for text boxes, which requires access to font metrics. These values must be provided by client tools. CSS requires left (and right) padding, margins, and borders to only apply to the first (and, respectively, last) boxes generated by an inline element when that inline element is split over multiple lines. Layout of inline boxes thus requires checking whether the box is the first or last box generated by its element, and applying margins, borders, and padding accordingly.

Opaque Boxes Many websites use features of CSS that are outside the subset that Cassius supports. To enable client tools to reason about these features, Cassius provides an escape hatch from its CSS semantics: an element can generate an opaque box, whose position is not related to any CSS properties or to any other boxes. Whether an opaque box floats is also not constrained. This lenient encoding of constraints on opaque boxes ensures that we allow any likely behavior of unknown CSS properties. Client tools can use assertions to constrain the behavior of these boxes as needed.

3.4 Restrictions on Floating Boxes

While it is possible for a declarative semantics to fully specify the behavior of floating boxes, such a specification would not be amenable to efficient automated reasoning. Because a floating box may affect the layout of every other box, and every block element can generate floating boxes, expressing these interactions in a quantifier-free logic would be prohibitively expensive. In particular, every float constraint in our semantics (such as Rule 1 in Section 3.3) would lead to $O(|\mathcal{B}|^2)$ constraints in a quantifier-free logic.

To enable efficient automated reasoning, the Cassius semantics imposes four restrictions on floating boxes, illustrated in Figure 7. The restrictions ensure that the layout of every box depends on the layout of the floating box closest to it in an in-order traversal of the box tree, rather than the layout of all floats. We do not believe these restrictions to be onerous, since the forbidden layouts can be often be achieved by adding extra elements to the document. Section 4 shows how Cassius exploits the floating box restrictions to asymptotically reduce the size of its SMT encoding. Without them, our declarative browser cannot solve the layout constraints for any of the benchmarks in Section 5.

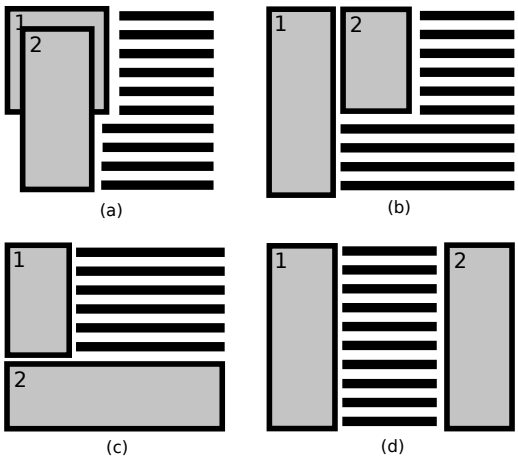

Figure 7: Interactions between floating boxes disallowed in Cassius. Each diagram is a possible layout of a document containing two floating boxes followed by text. In (a), the two floating boxes overlap, and the text must wrap around both. In (b), two floating boxes stack horizontally where the later box is smaller than the previous box. In (c), the second box is so wide it must wrap to the next line, but text continues to fit beside the first box. In (d), text flows around a left-floating and right-floating box. By disallowing these four types of interactions, text layout only relates a text box to its previous floating box. Note that each layout is achievable with Cassius; the HTML would have to be modified to rearrange the order of elements or add an extra block element around the text and the second div.

4. SMT Encoding

Cassius specifies the semantics of CSS layout declaratively, as a set of formulas expressed in the machine-readable SMT-LIB2 (Barrett et al. 2015) format. This specification closely parallels the description given in Section 3: quantified formulas define layout constraints on all boxes or all elements, with layout rules for different types of boxes encapsulated in functions. Such a high-level encoding has the advantage of being easy to audit for correspondence to the CSS standard. But it does not consistute a practical implementation of a declarative browser.

To provide a practical framework for building solver-aided layout tools, Cassius works by reducing the high-level semantics of a given declarative layout problem to the theory of quantifier-free linear real arithmetic (LRA). As described in Section 2, Cassius takes as input a (symbolic) stylesheet, a set of documents with their corresponding (symbolic) layouts, and, optionally, a set of assertions on the stylesheet and layout holes. These inputs are used to instantiate (or, ground) the quantifiers in the high-level semantics, yielding a set of quantifier-free LRA formulas. The resulting formulas are then simplified, fed to an off-the-shelf SMT solver (Z3 (De Moura and Bjørner 2008)), and the solver's output is used to either

fill the holes in the input or explain why a solution does not exist. We describe these steps in more detail below.

4.1 Grounding

In principle, it is easy to produce a quantifier-free encoding of the Cassius semantics for a given stylesheet C, a document D, and a layout B. We simply expand each universally quantified layout rule into a conjuction of ground (quantifier-free) constraints. For example, a rule of the form $\forall b \in \mathcal{B}, e \in \mathcal{E}, F(b, e)$ becomes a ground formula of the form $\bigwedge_{x \in B, y \in D} F(x, y)$, where x and y range over all boxes in B and elements in D, respectively. In practice, however, naive grounding produces a large formula that overwhelms the solver, especially when applied to float layout rules.

Cassius combats this encoding explosion by exploiting the float restrictions from Section 3.4, which ensure that the layout of every box can be computed from the layout of the preceding floating box in an in-order traversal of the box tree. The high-level semantics expresses this computation by specifying layout rules in terms of a float predecessor relation, is-flow-pred. For example, the following rule computes the top border edge (tbe) of a block box from the bottom border edge (bbe) of its in-flow predecessor:

$$\forall b_1, b_2 \in \mathcal{B}, \mathsf{is\text{-}flow\text{-}pred}(b_2, b_1) \wedge \mathsf{block\text{-}box}(b_1) \implies b_1.\mathsf{tbe} = b_2.\mathsf{bbe} + \mathsf{margin\text{-}gap}(b_2, b_1) \quad (2)$$

Instead of reducing such rules into equivalent ground formulas of size $O(|\mathcal{B}|^2)$, Cassius reduces them into equisatisfiable ground formulas of size $O(|\mathcal{B}|)$, by introducing a small set of *auxiliary uninterpreted functions*.

The key idea is to use the auxiliary functions to rewrite each high-level layout rule into a formula with at most one universal quantifier (over boxes). For example, pfs is an auxiliary function that maps each box to its float predecessor. Using pfs, Cassius rewrites our sample rule (2) into the following formula:

$$\forall b_1 \in \mathcal{B}, \mathbf{let}\ b_2 = b_1.\mathsf{pfs}\ \mathbf{in}\ \mathsf{block\text{-}box}(b_1) \implies b_1.\mathsf{tbe} = b_2.\mathsf{bbe} + \mathsf{margin\text{-}gap}(b_2, b_1) \quad (3)$$

The resulting formula is then simply ground into a conjunction of size linear in $|\mathcal{B}|$.

To ensure that this transformation preserves equisatisfiability, Cassius constrains the interpretation of each auxiliary function with suitable axioms. Thanks to a careful selection of auxiliary functions, these axioms can also be expressed using at most one universal quantifier. For example, pfs is constrained by the following axiom:

$$\begin{aligned}\forall b \in \mathcal{B},\ & \mathbf{if}\ \neg b.\mathsf{previous\text{-}sibling} \\ & \mathbf{then}\ b.\mathsf{pfs} = \mathsf{nil} \\ & \mathbf{elif}\ \neg \mathsf{floating\text{-}box}(b.\mathsf{previous\text{-}sibling}) \\ & \mathbf{then}\ b.\mathsf{pfs} = b.\mathsf{previous\text{-}sibling} \\ & \mathbf{else}\ b.\mathsf{pfs} = b.\mathsf{previous\text{-}sibling}.\mathsf{pfs}\end{aligned}$$

Thanks to the choice of auxiliary functions, the ground encoding as a whole—including both the transformed rules and the auxiliary axioms—is linear in $|\mathcal{B}|$.

The effectiveness of our grounding approach hinges on finding a small set of auxiliary functions that are both efficiently axiomatizable and sufficient to eliminate nested quantification (over boxes) from the high-level semantics. Our ability to find such a set for the Cassius semantics is a direct consequence of the float restrictions from Section 3.4. Intuitively, these restrictions define a subset of CSS that can be laid out with a *non-deterministic, incremental* version of the abstract layout algorithm (Figure 6) described in the CSS standard. The auxiliary functions introduced by Cassius encode the auxiliary information that the incremental algorithm would need to track in order to compute the layout in one pass, given a correct (angelically chosen) order in which to traverse the tree of boxes. Our grounding approach can thus be understood as encoding an arbitrary execution of this incremental algorithm on a specific (symbolic) stylesheet, documents, and layouts.

4.2 Constraint Simplification and Solving

In the final stages of declarative layout, Cassius simplifies the ground encoding of the semantics using a custom formula optimizer, and passes the resulting constraints to Z3. The optimizer avoids simplifications that interfere with unsatisfiable core extraction, and embeds metadata into the encoding (using Z3's named constraints) that relate the optimized and ground semantics. This information is used to lift the solver's output into a solution to the declarative layout problem received by Cassius.

5. Evaluation

The previous sections described Cassius, a formal specification of CSS semantics with an efficient encoding in the theory of quantifier-free linear real arithmetic. This section demonstrates that the Cassius semantics faithfully models the CSS standard; that it is rich enough for practical use; and that its encoding to LRA is efficiently-solvable, enabling easy creation of semantic tools for CSS that work on fragments of real web pages.

5.1 Correctness of the Cassius Specification

To ensure that Cassius correctly captures the W3C CSS 2.1 standard, we compared it to Mozilla Firefox on a set of 2075

standard conformance tests (CSSWG 2011) for imperative browsers. Two experiments were done. In the first experiment, we checked that the Cassius semantics for CSS accepts the rendering produced by Firefox, demonstrating that this semantics is *sufficiently weak* (with respect to Firefox). Our results show that Cassius agrees with Firefox on all but six tests, on which Cassius produces correct layouts according to the standard, while Firefox produces slightly different layouts due to rounding error. In the second experiment, we checked that the Cassius semantics for CSS rejects other renderings by using a form of solver-aided mutation testing. Only 0.7% of mutants are accepted by Cassius, largely due to font metrics, which Cassius does not model. These results show that the Cassius semantics is *sufficiently strong* in practice, despite our liberal modeling of line breaking (Section 3.3). We describe our experimental setup and results below.

Methodology The official conformance tests (CSSWG 2011) measure the interoperability and correctness of CSS layout implementations. Tests exist for every section of the CSS standard, including many aspects of CSS layout (such as fonts, colors, or print media) not described by Cassius. The Cassius fragment of the standard is covered by 2075 conformance tests. Each test (e.g., Figure 8) consists of a web page with an English-language description of the expected output. Tests also have an associated reference page that achieves the expected layout using a simpler stylesheet. The tests are small, rarely consisting of more than a dozen elements and a few stylesheet rules. Furthermore, the W3C maintains a public website through which volunteers manually compare a browser's rendering of the test to the instructions and to the reference page.

5.1.1 Acceptance Tests

We applied Cassius to all 2075 relevant conformance tests. To check that it produces the expected renderings, we use the Mozilla Firefox 41.0.1 browser as a test oracle, since volunteers have already checked that Firefox passes these tests. For each test, we employ a script (based on the CSSOM JavaScript API) to extract the layout generated by Firefox. This fully-concrete layout, plus the document tree and stylesheet, are fed to Cassius to check that its specification allows the given rendering. Note that such a check exercises our encoding even with fully concrete inputs: Cassius uses the solver to search for a trace of its non-deterministic layout algorithm that admits those inputs.

Results The results are shown in Table 1. The tests are grouped by the section of the standard that they cover. Tests were run on an Intel Core i7-4790K quad-core CPU, with a version of Z3 built from the opt branch on 11 March 2015. Of the 2075 tests, Cassius agrees with Firefox on all but six; the disagreements are described below. The full suite of tests took 138 minutes to run, for an average of less than three seconds per test.

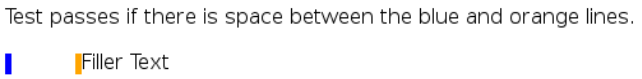

Figure 8: A test from the W3C CSS 2.1 conformance tests; this one is named padding-left-applies-to-008. Tests pass or fail based on the English-language instructions provided on each page or by comparison to a reference page that achieves the desired layout using a different, simpler stylesheet.

CSS Standard	Test group	Accept	Reject
§8.1	Box dimensions	2	0
§8.3	Margins	218	0
§8.3.1	Collapsing	12	1
§8.4	Padding	274	1
§8.5.1	Border width	278	2
§8.5.2	Border color	734	0
§8.5.3	Border style	71	0
§8.5.4	Border interactions	113	0
§8.6	Inline	2	0
§9.2.2	Inline boxes	2	0
§9.4.1	Block formatting	14	0
§9.4.2	Inline formatting	25	0
§10.2	Width	70	1
§10.3.1	Inline non-replaced	3	0
§10.3.3	Block non-replaced	6	0
§10.4	Min-/max-width	67	0
§10.5	Height	64	1
§10.6.1	Inline non-replaced	1	0
§10.6.3	Block non-replaced	5	0
§10.6.7	Flow roots	1	0
§10.7	Min-/max-height	66	0
§10.8	Line height	1	0
§10.8.1	Leading	39	0

Table 1: The results of applying Cassius to the W3C test suite. The "Accept" and "Reject" columns give the number of tests on which Cassius does and does not accept Firefox's rendering. See the text for a description of the cases where Firefox and Cassius disagree.

Disagreements Between Cassius and Firefox On six tests, Firefox and Cassius disagree. The cause of the disagreement is rounding error within Firefox, and Cassius's rendering is correct according to the CSS standard.

There are two sources of rounding error in Firefox. First, Firefox represents on-screen lengths as fixed-point values rounded to a sixtieth of a pixel. This accuracy is usually sufficient to render web pages properly, since errors of sixtieths of a pixel are not visible. In some cases, the true position of a point on the screen is not an exact multiple of a sixtieth of a pixel; for example, the test margin-collapse-032 creates a box whose padding is 2% of 1898 pixels, that is, $37.96 = 2277.6/60$ pixels. Firefox rounds the padding

of the box to 37.95 pixels. Since Cassius uses real (infinite-precision) arithmetic, per the CSS standard, its specification does not admit this rounded value, thus disagreeing with Firefox. Second, Firefox rounds border widths and text positions to the nearest pixel, to ensure that text can be properly hinted and that borders appear to have the same width on both sides of any box. Modifying each test to correct the rounding errors results in a passed test.

5.1.2 Rejection Tests

To check that Cassius rejects invalid renderings, we mutated the suite of 2075 acceptance tests as follows. For each test, we randomly chose a block box's width, height, or x or y position, and replaced it with a hole. Cassius was then asked to find a rendering of the page that differs from Firefox's in the chosen measurement. Finding such a rendering represents a failure—a test case on which the Cassius semantics is weaker than that of Firefox. This procedure was repeated ten times for each test in the test suite, resulting in 20750 total attempts to find an invalid rendering accepted by Cassius.

Results Of the 20750 mutants, Cassius accepted only 152, for an overall success rate of 99.3%. We manually investigated every accepted mutant, and found that they were all accepted due to two causes. The majority of the mutants (126 out of 152) were accepted because Cassius does not model font metrics. In particular, Cassius accepts renderings that move the boundary between two lines of text. An imperative browser rejects these renderings by computing the A and D font metrics, which determine the height of a line of text. The remaining mutants (26 out of 152) are due to non-determinism in the CSS 2.1 algorithm for computing the shrink-to-fit width of floating boxes. Cassius treats the non-determinism in the specification by accepting several renderings, allowing Cassius to sometimes find renderings different from the one found by Firefox. Several drafts for the CSS 3 standard fully specify the shrink-to-fit algorithm, so Cassius could be updated to remove this non-determinism once the draft standards are accepted.

5.2 Practicality of the Cassius Specification

Tests against the W3C test suite show that Cassius faithfully formalizes its fragment of the CSS semantics, closely matching the Firefox implementation. To demonstrate that this fragment is practically useful, we applied Cassius to the layout of five large, popular websites: a Google results page; a Wikipedia article; the Yahoo! home page; a Baidu results page; and the Amazon home page. Our results show that all of these web sites are expressible in the Cassius fragment of CSS, with minor preprocessing performed by client tools.

Methodology We performed four experiments on each of our five sample websites: checking whether Cassius accepts the Firefox's rendering of the website; verifying the website against a desired property; debugging the Firefox layout with respect to an undesired property; and synthesizing a stylesheet for the website from a sketch.

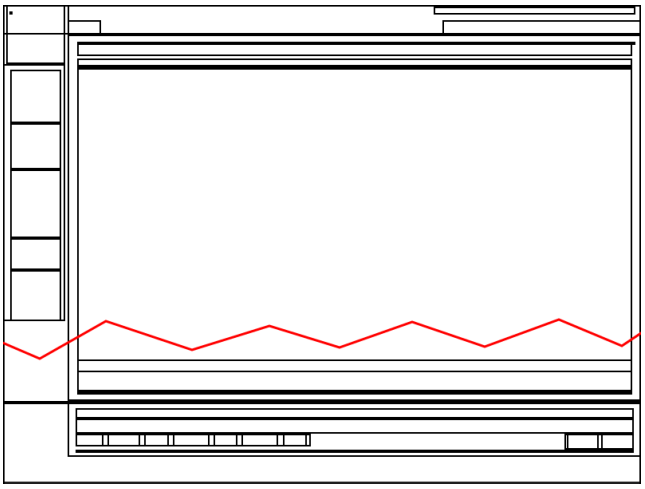

Figure 9: The truncated Wikipedia page used in our benchmarks. Each rectangle is one box; many overlap. The boxes show the overall design of the page, shared across Wikipedia's articles. The sharp red line marks where a segment of the rendering, containing the featureless middle of the image, was removed.

As for the W3C conformance tests, we used Firefox to render each page, automatically capturing the resulting layout and all applicable stylesheet rules. We then used a simple preprocessor to automatically convert the extracted layout, CSS, and HTML into a declarative layout problem accepted by Cassius. Our preprocessor ignores unsupported CSS properties,[7] and it treats unsupported boxes (such as tables) as opaque. Unsupported selectors are treated as a disjunction of identifier selectors: the preprocessor gives every matching element a unique identifier, and it creates one copy of the selected rule for each matched identifier. The preprocessor also replaces all boxes below a given depth in the layout tree with opaque boxes, effectively truncating the page. This truncation is necessary for Cassius to terminate within an hour. We chose the depth parameter manually so that the truncated page retains the large-scale structure of the corresponding benchmark page, while enabling faster layout times; Figure 9 shows an example of a truncated page. Finally, rounding errors due to Firefox were manually corrected.

5.2.1 Acceptance Tests

For each benchmark page, Cassius admits the rendering made by Firefox. Detailed results are given in Table 2, including the running time of Cassius, the size of the layouts and stylesheets reasoned about, and the size of the constraints generated by Cassius. The conformance checks for these pages are slower

[7] Except the `max-width` property, which replaced the `width` property when set.

	Elt	Box	Rule	Size	Time
Amazon	18	54	16	23k	14.9+2.8s
Baidu	38	51	15	178k	11.9+14.1s
Google	32	35	17	158k	11.5+3.7s
Wikipedia	45	50	30	215k	13.9+42.5s
Yahoo!	40	39	26	263k	46.6+9.0s

Table 2: Accepting the rendering of five popular websites with Cassius. The "Elt", "Box", and "Rule" columns count the number of non-content elements and applicable CSS rules, with duplicate rules (due to unknown selectors) counted once. "Size" counts the number of expressions and declared variables in the SMT problem. The "Time" column is broken into time to generate and solve the constraints.

than for the W3C tests, but they still take only a few seconds each.

5.2.2 Tool Building atop Cassius

To demonstrate the utility of Cassius as a tool-building framework, we used it to prototype the three semantic tools for CSS described in Section 2: a verifier, a debugger, and a synthesizer. By building on top of Cassius, all three tools took just four days to implement, requiring less than 300 total lines of code. We evaluated each of them on the five websites described above. Our results show that, even with our modest implementation effort, these tools work on fragments of real websites, producing results in a matter of minutes.

1. To evaluate verification, we measured the time needed to verify that no text boxes or links (`<a>` elements) are overlapped on a website, for any viewport size between 800 and 1920 pixels.
2. To evaluate debugging, we paired each website with an unsatisfiable assertion negating the position and size of randomly-chosen boxes. We passed these pairs to the debugger and measured the running time, the size of the resulting unsatisifiable core, and the number of CSS properties identified as relevant.
3. To evaluate synthesis, we replaced the bodies of 1–25 randomly-chosen rules with expression holes and measured the time to complete the resulting sketches.

Results Verification of each site took between 2 and 12 seconds (Table 3). Debugging produced small cores and identified few CSS properties: debugging time ranged from 1–5 seconds and identified cores of 2–7 CSS rules and 4–10 CSS properties (also in Table 3). Synthesis time was substantially longer, but it grew slowly with the number of holes (see Figure 10), with the fastest stylesheets synthesizing 25 expression holes in minutes. Note that the Amazon stylesheet is particularly difficult for Cassius to synthesize. We suspect this is due to the inclusion of selectors that match many elements, which makes the rule bodies of those selectors harder to synthesize.

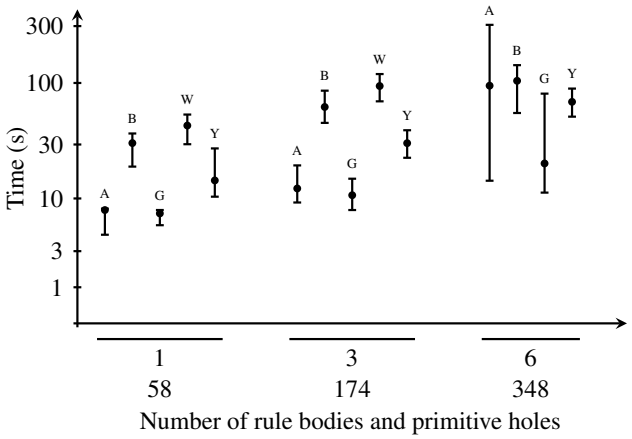

Figure 10: Each bar represents ten random sketches of one website (labeled above), with a fixed number of expression holes. Minimum and maximum outliers are removed and the median is labeled with a thick dot. A 5-minute timeout was used for all experiments, and all sample sets that included timed-out runs are discarded and not shown.

	Verify	Debug	Rules	Properties
Amazon	4.9s	0.7s	1	2
Baidu	3.7s	3.2s	5	6
Google	5.7s	2.7s	3	3
Wikipedia	2.0s	0.7s	3	4
Yahoo!	12.2s	5.5s	1	1

Table 3: Debugging and verification time for five popular websites using the Cassius prototype tools. The debug time lists just constraint solving time; constraint generation time identical to that in Table 2.

5.3 Scalability of Cassius in the Presence of Floats

The tests with real-world website fragments and the W3C conformance tests show that Cassius produces constraints that can be solved quickly, with verification, debugging, and synthesis. But none of these benchmarks makes heavy use of floats. To evaluate the scalability of our encoding in the presence of floats, we generated 14 web pages with a variable number of elements by changing the number of products in the running example (Figure 1). The resulting web pages contain a simple CSS stylesheet of three rules with expression holes for their bodies (plus two rules from Firefox's browser stylesheet), and an HTML document with 0–13 floating boxes. Cassius was asked to fill each stylesheet sketch, and the results are graphed in Figure 11. Because almost all elements float, synthesis is slower than for the websites from Section 5.2. However, it still completes within a few minutes. Without our encoding optimizations based on float restrictions, none of the benchmarks in this paper (including the small W3C tests) complete within an hour.

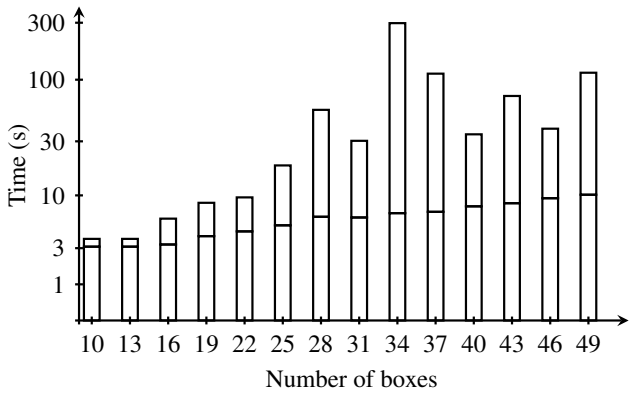

Figure 11: Synthesis time for increasing sizes of the running example document (Figure 1). The break in each bar is constraint generation time; the top is total time. The unpredictable nature of Z3 performance explains the variance in solve time. Constraint preparation time increases smoothly and linearly, while solve time increases unpredictably but also slowly.

5.4 Ease of Extending Cassius

Over the course of preparing the Cassius semantics and evaluating it on the websites from Section 5.2, we extended Cassius multiple times to support new CSS properties. Cassius's support for the text-align, overflow, position, left/right/top/bottom, white-space, and box-sizing properties were all added with minimal changes to the semantics and with a few hours required to plan, implement, debug, and test each extension. This experience suggests that future extension of Cassius can be done quickly, efficiently, and without large-scale modification of the Cassius semantics.

6. Related Work

Cassius builds on a foundation of work in solver-aided languages, and was inspired by work on web development tools in the HCI community. The formalization of CSS further builds on work in parallelizing web browsers.

Solver-aided Languages Solver-aided languages (Torlak and Bodik 2013) use SAT and SMT solvers to automate programming tasks such as verification, debugging, and synthesis of code. Boogie (Leino 2008) serves as a back-end for general purpose program verification by using Z3 to prove verification conditions (Leino 2010). Alive (Lopes et al. 2015) and PEC (Kundu et al. 2009) verify compiler optimizations using an SMT solver. Batfish (Fogel et al. 2015) verifies the dataplane for Datalog programs. In each case, verification conditions and program properties of a high-level language are compiled to efficiently-solvable SAT or SMT constraints. Cassius follows a similar architecture for CSS: queries about CSS stylesheets, including synthesis and debugging queries, are compiled to SMT constraints and solved with Z3. Smten (Uhler and Dave 2014) and Rosette (Torlak and Bodik 2014) are solver-aided *host languages*—they translate a programming language interpreter into a solver-aided language backend. Unlike these general-purpose languages, which aim at automating a wide spectrum of problems, Cassius focuses on providing both a declarative semantics for CSS and an efficient host platform for automating web page layout.

Constraints in Web Design Using constraint solving for layout has a long history, and applications to web page layout, as a replacement for CSS, have been proposed (Badros et al. 1999). Like these techniques, Cassius views CSS files as a set of constraints that are solved to lay out the page. Unlike previous work, however, Cassius does not replace CSS or modify existing browsers. Instead, it provides a mechanized semantics for CSS, and as such, it could be used to compile constraint-based layout languages to CSS.

Web Development Tools Several tools for automating aspects of web development have been proposed, especially within the human-computer interaction (HCI) community. Bricolage (Kumar et al. 2011) uses heuristics to transfer the design of one web page to another; this requires matching the nodes on two web pages with similar content, and then transferring the stylesheet from one web page to another. Webzeitgeist (Kumar et al. 2013) uses similar techniques to organize a corpus of common design elements across web pages. SeeSS (Liang et al. 2013) automatically tracks changes in web page layout due to changes in CSS, to help developers avoid introducing bugs as they modify CSS code. Further afield from web development, ReVision (Savva et al. 2011) applies machine vision to understand the structure of data visualizations, and synthesize new visualizations of the same data; similar tools (Harper and Agrawala 2014) allow easy modification of data visualizations built with the D3 library. Remaui (Nguyen and Csallner 2015) uses machine vision, optical character recognition, and heuristics to synthesize Android application layouts from mock-ups. Tools have also been built to identify redundant CSS rules, soundly accounting for rules that are only triggered by JavaScript modifications to a web page (Hague et al. 2014). Cassius provides automation facilities that complement these efforts, and investigating avenues for integration is an interesting direction for future work.

Parallel Web Browsers Previous work (Meyerovich and Bodik 2010) has encoded a partial specification of CSS as an attribute grammar. This specification is then used to generate parallel execution schedules for web page layout. Attribute grammars have also been used to synthesize data visualizations (Hottelier et al. 2014), where the synthesis selects between possible constraints on an element in the visualization. Unlike prior formalization efforts, Cassius specifies CSS semantics in the theory of (quantifier-free) linear-real-arithmetic. The main advantage of this encoding is the ability to easily add constraints that do not follow the

tree structure of the document, such as the specification of floating elements, and to leverage off-the-shelf SMT solvers for automated reasoning.

7. Conclusion

This paper presents Cassius, a new solver-aided framework for automated reasoning about web page layout. Cassius includes a declarative specification of a substantial fragment of the CSS semantics, along with an efficient encoding of that specification in the theory of quantifier-free linear real arithmetic. This encoding exploits the observation that the Cassius fragment of CSS is amenable to layout with a non-deterministic incremental algorithm, whose semantics can be expressed with a linear (rather than quadratic) number of constraints. We demonstrate the utility of Cassius by using it to prototype three solver-aided tools for CSS—a verifier, debugger, and synthesizer—with under 300 lines of code. Thanks to the efficiency of the Cassius encoding, our prototypes run in minutes on fragments of real-world stylesheets and documents.

Acknowledgments

We the anonymous reviewers for guidance and valuable suggestions while preparing the final version of this paper. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1256082.

References

G. J. Badros, A. Borning, K. Marriott, and P. J. Stuckey. Constraint cascading style sheets for the web. UIST'15, 1999. doi: 10.1145/320719.322588. URL http://doi.acm.org/10.1145/320719.322588.

C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

CSSWG. CSS2.1 test suite, 2011. URL http://test.csswg.org/suites/css2.1/20110323/html4/toc.html.

L. De Moura and N. Bjørner. Z3: An efficient SMT solver. TACAS'08/ETAPS'08, 2008. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. NSDI'15, Oakland, CA, May 2015. USENIX Association.

M. Hague, A. W. Lin, and L. Ong. Detecting redundant CSS rules in HTML5 applications: A tree-rewriting approach. *CoRR*, 2014. URL http://arxiv.org/abs/1412.5143.

J. Harper and M. Agrawala. Deconstructing and restyling d3 visualizations. UIST '14, 2014. doi: 10.1145/2642918.2647411. URL http://doi.acm.org/10.1145/2642918.2647411.

T. Hottelier, R. Bodik, and K. Ryokai. Programming by manipulation for layout. UIST'14, 2014. doi: 10.1145/2642918.2647378. URL http://doi.acm.org/10.1145/2642918.2647378.

R. Kumar, J. O. Talton, S. Ahmad, and S. R. Klemmer. Bricolage: Example-based retargeting for web design. CHI'11. ACM, 2011. doi: 10.1145/1978942.1979262. URL http://doi.acm.org/10.1145/1978942.1979262.

R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton. Webzeitgeist: Design mining the web. CHI'13, 2013. doi: 10.1145/2470654.2466420. URL http://doi.acm.org/10.1145/2470654.2466420.

S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. PLDI'09, 2009. doi: 10.1145/1542476.1542513. URL http://doi.acm.org/10.1145/1542476.1542513.

K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, June 2008. URL http://research.microsoft.com/apps/pubs/default.aspx?id=147643.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR'10, 2010. URL http://dl.acm.org/citation.cfm?id=1939141.1939161.

H.-S. Liang, K.-H. Kuo, P.-W. Lee, Y.-C. Chan, Y.-C. Lin, and M. Y. Chen. Seess: Seeing what i broke – visualizing change impact of cascading style sheets (CSS). UIST '13, 2013. doi: 10.1145/2501988.2502006. URL http://doi.acm.org/10.1145/2501988.2502006.

N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'15, 2015. doi: 10.1145/2737924.2737965. URL http://doi.acm.org/10.1145/2737924.2737965.

L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. WWW'10, Raleigh, North Carolina, USA, 2010.

T. A. Nguyen and C. Csallner. Reverse engineering mobile application user interfaces with remaui. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'15. IEEE, Nov. 2015.

M. Otto and J. Thornton. Bootstrap: the world's most popular mobile-first and responsive front-end framework, 2015. URL http://getbootstrap.com/.

M. Savva, N. Kong, A. Chhajta, L. Fei-Fei, M. Agrawala, and J. Heer. Revision: Automated classification, analysis and redesign of chart images. In *ACM User Interface Software & Technology (UIST)*, UIST'11, 2011. URL http://idl.cs.washington.edu/papers/revision.

A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. ASPLOS XII, 2006. doi: 10.1145/1168857.1168907. URL http://doi.acm.org/10.1145/1168857.1168907.

E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, PLDI'14, 2014.

R. Uhler and N. Dave. Smten with satisfiability-based search. Oct. 2014. doi: 10.1145/2714064.2660208. URL http://doi.acm.org/10.1145/2714064.2660208.

W3C. Css basic box model, Aug. 2007. URL http://www.w3.org/TR/css3-box.

W3Techs. Usage statistics of CSS for websites, Nov. 2015. URL http://w3techs.com/technologies/details/ce-css/all/all.