Fomal Verification of Dynamic Web Pages

Skyler Griffith
University of Utah

UUCS-21-006

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

22 April 2021

Abstract

Web pages are widely used and error prone pieces of software, making them opportune for formal verification techniques. However, much of the prior literature, such as a project called Cassius, focuses on verifying static pages while largely ignoring dynamic ones. This presents a gap in the research, as the vast majority of the modern web is made up of dynamic pages. In this project we expand upon Cassius to show that it can also work on dynamic pages. We take a simple dynamic page with a growing list and build havoc and induct tactics into Cassius' proof framework to give it the power to formally verify facts about the dynamic page. In addition to successfully verifying a dynamic web page, we also considerably improve Cassius' performance on large static pages.

FORMAL VERIFICATION OF DYNAMIC WEB PAGES

by

Skyler Griffith

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science

In

Computer Science

Approved:

Pavel Panchekha Thesis Faculty Supervisor	Mary Hall Director, School of Computing

Jim de St. Germain
Director of Undergraduate Studies

April 2021

ABSTRACT

Web pages are widely used and error prone pieces of software, making them opportune targets for formal verification techniques. However, much of the prior literature, such as a project called Cassius, focuses on verifying static pages while largely ignoring dynamic ones. This presents a gap in the research, as the vast majority of the modern web is made up of dynamic pages. In this project we expand upon Cassius to show that it can also work on dynamic pages. We take a simple dynamic page with a growing list and build havoc and induct tactics into Cassius's proof framework to give it the power to formally verify facts about the dynamic page. In addition to successfully verifying a dynamic web page, we also considerably improve Cassius's performance on large static pages.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Millions of people use the web for a variety of reasons, from visiting the personal page of a university professor to securing health insurance on `healthcare.gov`. In addition, web pages are used in many different settings: on a computer, a smart phone, or even a television. Each of these different use cases introduces a host of challenges and opportunities for bugs.

A single web page on a single device can also be loaded in a near countless number of configurations, as nearly every modern browser includes the ability to resize the window or zoom in and out. It is possible, and even fairly common, for a new configuration of the browser, an unexpected platform, or a combination of the two to introduce bugs that break the functionality of a web page [3].

To add to all of this, web pages are complex. They are generally programmed in 3 different languages: HTML, CSS, and JavaScript. The code in these languages must be well formed and interact with the rest of the page's code correctly. Given all the ways that web pages are used, and all the ways they can break, there is an obvious utility to having a tool that could be used to prove that a page is bug free.

There are two parts to this task. First, you need a way to specify the expected behavior of the page, in terms of its appearance and functionality. Second, you need a tool that takes the HTML, CSS, and JavaScript of the page and checks that it always meets that expected behavior for all possible configurations.

A system called Cassius already accomplishes this for *static* web pages. Its tactic language makes it possible to both specify the expected appearance of a web page and to prove theorems about that appearance for a specific page. Cassius works by constructing a set of queries that are fed into the SMT solver Z3 [2]. Specifically, these queries are the *negation* of the specified appearance. In other words the query states that "for our web

page, this specification is violated by some configuration". If Z3 finds a configuration of the web page that satisfies that query, then the user is given an example of how to trigger a bug, which they can use to improve the page. If it cannot then the web page satisfies the specification.

The problem with this system is that it only works on static web pages, which make up a small subset of the modern internet, because it views web pages as a snapshot of the page's current state. This means that to verify something with multiple states, such as a dynamic page, Cassius has to go through each state and verify them one by one. A dynamic web page is capable of having an infinite number of states, making this process potentially interminable on dynamic web pages. This is not an unsolvable problem however, and we can expand this system to show that it is possible to verify something about dynamic pages.

To verify a given dynamic page we will need to use the Cassius tactic language to create a proof that can be true for all possible states of the page. This will involve determining how the JavaScript changes the pages state, and then using techniques like havoc and induction to prove that a theorem is true no matter how many times a given part of the state is changed.

To accomplish all of this we must first catalog all the changes the page's JavaScript can cause. To do this we treat JavaScript as a list of changes that happen to the web page. We use a JavaScript parser called Esprima [1] to help us convert JavaScript code into a list of changes made to the page.

Once we have the ability to analyze what changes a page's JavaScript is capable of making to the page's state we need to be able to show that those changes happening won't change the outcome of the proof. This requires that we add new tactics to the pre-existing proof framework in Cassius. To better explore this method we use an example page's as a proof of concept that the ideas discussed so far are feasible. We create a simple web page with a list that gets a new element every time the user presses a button. The JavaScript is analyzed and we get the set of changes that can happen to the state of the page using the process described above.We can then start focusing on what changes need to be made to the proof such that it will be able to take the changes to the page into account.

There are two things that change across states for the page: the text in each element

of the list, and the length of the list. Our proof will need to be capable of taking both these changes into account. To accomplish this we add a `havoc` and `induct` tactic to the pre-existing proof structure that Cassius has.

First, we need to take changes to the text value of the elements of the list into account. To do this we leverage the way in which Cassius sees values in a page's elements as proof constraints. By removing a specific value we remove a constraint on the Z3 query associated with that element. This means we have made the result of the proof the same no matter what that value might be, and to support all of this we have added the ability to `havoc` values in the Tactic Language.

Next is the problem of proving the theorem true no matter the length of the list, which lends itself to a proof by induction. The only problem is that Z3 can't handle inductive logic, so we have to create a system that does a lot of that leg work. What Z3 *can* do is verify all the different cases of an inductive proof, so we make Cassius change the list into several different states that can be used as base case and inductive step of an inductive proof. We then use this technique to prove theorems about the list by induction, and by extension prove theorems about the page no matter the length of the list.

After making these changes we ran the system with our proof on the web page and it fully verified, providing a proof of concept for verifying dynamic pages. It also had the side benefit of improving Cassius' performance on static pages. Induction allows the user to verify a fact about a long list of items with a much shorter Z3 query, making the entire system better at handling static pages.

CHAPTER 2

BACKGROUND

Before we explore the details of how we expand upon Cassius, let us establish how Cassius verifies static pages. From the user's perspective there are two major pieces to Cassius: the capture script and the tactic language.

The capture script exists to take a snapshot of the web page the user wants to prove something about. It takes a single web page and converts it into an intermediate representation (IR) that is a series of s-expressions describing things about the page's fonts, stylesheet, browser, HTML, and layout. This information is later used to formulate a Z3 query that establishes many of the preconditions to a user's proof.

Next, the tactic language allows the user to prove things about the captured page. This language is designed to allow a user to write proofs of theorems about specific web pages that Cassius can then take and formally verify. To support this the tactic language supports the creation of 3 objects: pages, theorems, and proofs.

A page is an object that is used to define the web page that is being verified. To create it one must first take a snapshot of the web page and save it using the capture script. This info can then be loaded into the proof with the `load` tactic, which will create a page object that Cassius can perform a proof on.

A theorem is an object that is used to define some fact the user wants to prove about a web page. Cassius ships with a set of common theorems that the user can bring into their proof, or they can write their own. Generally these facts will be something along the lines "this bug does not exist on this page," for example, the theorem that we later explain further in Chapter 4 is that the top of a list will always be above the bottom of that list.

Finally, the proof object behaves like a function that takes in one theorem and one or more pages and contains the steps that will be taken to prove that the theorem is always true for all of the provided pages. The user can state what these steps are using tactics

established in the language. A clear example of how that works in detail can be found in Chapter 4.

CHAPTER 3

RELATED WORK

This project relies heavily on work previously done on the Cassius project. In addition to Cassius it also builds upon ideas explored in a project called Cornipickle. Here a summary of both projects is provided.

Cassius is a web browser developed specifically to assist with formal verification on web pages. It allows the user to capture the HTML and CSS of their web page with a script that converts that information into something that Cassius can use. It then allows the user to write a specification in a Lisp-like proof language, in which the user points to captured versions of the web page, writes theorems that define what they wish to prove, and creates proofs that verify those theorems on the page.

Cassius [6, 7] then takes the captured page and specification and turns it into a query it can feed into Z3. It leaves certain facts about the configuration of the page unspecified, which allows it to create a much more general proof of the web page. Before verification some pre-processing on the query is required. The query produced by combining the spec and the captured page forms something like $\forall W, P(W)$ where W describes the possible configurations of the web page and P is the property that spec describes. Unfortunately Z3 cannot understand $\forall$-quantified statements, so the statement is negated to become $\exists W, \neg P(W)$. If Z3 returns UNSAT for this query then the web page meets the spec for all configurations. If it returns SAT it will provide a configuration that satisfies the query, and by extension breaks the provided specification.

The size of the Z3 queries produced by a simple web page are large, however, and does require that this system be better optimized. To handle this Cassius takes advantage of the tree-like structure of a web page to modularize the proof [5]. Instead of proving every property of the page across all possible pages, it proves some intermediate property for each component of the web page. Since components are smaller, those queries are easier.

Cassius then proves the theorem provided in the spec based on the intermediate properties previously proven for each of the components. This leads to several small queries instead of one very large one, which proves to verify much faster.

One other weakness Cassius has is that, because it is proving facts on a captured snapshot of the web page, it is incapable of verifying dynamic pages. This proves to be a rather major problem, since modern websites almost always use JavaScript. This is where one alternative tool, Cornipickle [3], proves useful. Cornipickle provides a specification language you can use to define how the HTML, CSS, and JavaScript should behave on a page. It defines the JavaScript aspec by allowing the proof language to state what changes the JavaScript code will cause on the page and ensure that if those changes happen then the page still meets the specification. There is one weakness to Cornipickle, however, and that is that its specifications only verify a specific, given configuration of a web page. This contrasts with Cassius's ability to verify all possible configurations, and leaves us with a proof that is weaker.

CHAPTER 4

CASE STUDY

As mentioned in Chapter 1 in this project we create a specific page as a proof of concept for verifying dynamic pages. Before we talk about what we need to change about Cassius to make it capable of verifying that dynamic page, let us establish the details of our proof of concept.

4.1 The Web Page

First, let's focus on the web page, which consists of a list and a button. The list starts empty when the page is initially loaded, and gets one element longer when the button is pressed. The HTML that describes the list and button in the initial state of the page are as follows.

```
<!doctype html>
<html>
<body>
  <ul id="list"></ul>
  <button type="button" id="click" onclick="add_element()">
    Click Me!
  </button>
</body>
</html>
```

When combined with this script:

```
function add_element(e) {
    var list = document.querySelector("#list");
    var elt = document.createElement('li');
    var id = (list.children.length + 1)+'/'+window.iteration;
    elt.id = 'element-' + id;
    elt.innerText = 'value-' + id;
    list.appendChild(elt);
}
```

and Firefox's default stylesheet [4] we have a simple web page that has an unbounded number of states, those being the different possible lengths of the list. We will use this page as the basis of our proof going forward.

4.2 Verifying JavaScript

Now that we have built the page, we need start verifying it. Before we can actually prove anything about the page we need to take into account what the JavaScript does to the page. Creating a tool that verifies arbitrary JavaScript code is well outside the scope of this thesis, so instead we think about JavaScript as a list of potential changes that happen to the page's state each time the JavaScript runs.

To support this the capture script is expanded to contain a JavaScript parser called Esprima [1]. It takes the script embedded in the web page and uses the parser to convert the script into a list of changes like the one seen in Figure 4.1. This IR describes what the script does in a way that Cassius can actually understand and work with.

Next, Cassius takes the IR of the script and the page, applies the changes described in the IR, and updates the page to bring it to the state it would be at after the script ran once. In this instance a single child is appended to the end of the list. We assume that these can be repeated any number number of times, so we now know that the list has an unknown number of children. Now we need to decide on a theorem to prove, and find a way to prove it no matter how many times the change we just performed happens.

4.3 Proof Theorem

After the page is built and the JavaScript taken into account we begin our proof. To start, we need to decide on a theorem T about the dynamic section of the page, in this case the list, that we want to prove. For this example we will prove that the top element of the list is above the bottom element of the list, like so.

```
(theorem (top-above-bottom b)
  (>= (bottom b) (top b)))
```

Here, `(top b)` refers to the y-coordinate of the top edge of the box `b`, and `(bottom b)` refers to the y-coordinate of the bottom edge of the box `b`. Since in a web page the Y-axis

points down and the X-axis to the right, to be above something means to have a smaller Y coordinate. Therefore saying the top of `b` is less than the bottom of `b`, as the theorem above does, means that the top is above the bottom. While this may seem trivially true, there are certain behaviors, such as negative margins, that make it possible for this theorem to be false on a given web page.

4.4 Components

Now that we have a page and a theorem we wish to prove, we need to figure out how to go about proving the theorem T. In Cassius, proving a theorem means that the theorem is true for all possible configurations of the given web page, where configuration could mean different size of the browser, different zoom level, or in this example a different state for the dynamic portion. In order to avoid an extremely large and slow Z3 query, Cassius breaks down the web page into base parts, and checks that the theorem is true no matter how those parts are put together.

Web page design simplifies breaking down a page into pieces through the box and element trees. The central data structure behind every web page is a tree, where the root is the browser that the page is being drawn on, and each element on the page, such as the list, is a child of either the root or a different element. For an example see Figure 4.2. We will from here on out refer to the nodes of this tree as components, and will formulate our proof around them.

4.5 Proof walk through

The full proof can be seen in Figure 4.3, but let us walk through it here. First we use the `component` tactic to define the components that we will be focusing our efforts on: the list, the button, and the elements inside of the list that are called todos. These are the things that can move around so they're what our proof is going to focus on.

Next, the `havoc` tactic is used on the text and width values in the different todos. The purpose of the `havoc` tactic is to cause Cassius to "forget" that the individual todo items have different text inside of them. That makes all the todo items identical, which helps us meet the preconditions for an inductive proof later, which is described in Section 5.3. The technical details of how this works are further explored in Section 5.2, but the short version

```
(script add_element
  (let list (select (id list)))
  (let elt (create [li]))
  (set elt ':id ?)
  (set elt ':text ?)
  (append-child list elt))
```

Figure 4.1. The Cassius capture script IR of the JavaScript from our page. This describes a single addition to the list in our web page.

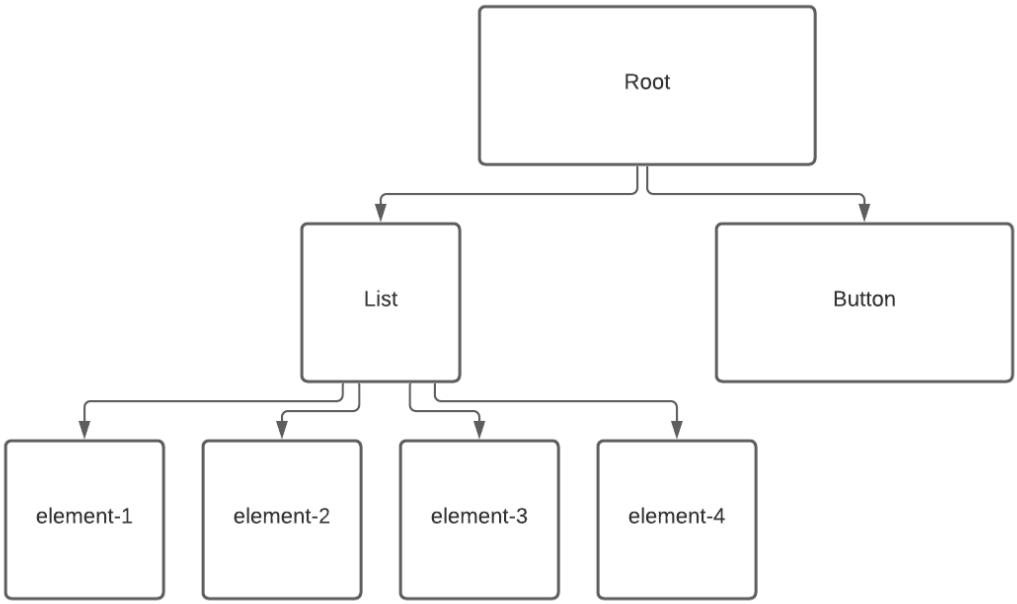

Figure 4.2. Above is a tree representation of our web page, where each node represents a single component of the page. The Cassius tactic language lets us define facts about specific components and localize our proof to different nodes or groups of nodes in the page's element tree.

is this: the text in each of the todos is a constraint on the proof, and by removing the text we remove that constraint, allowing the proof to reach the same result no matter what the text is.

Next, we use the `assert` tactic to verify the following CSS details: that the elements of the list always have a positive height, that they don't have any floats tracked, and that they don't have any negative margins. After these are verified we can establish the precondition that the list also doesn't have any floats tracked using the `pre` tactic. When these are all verified, the final preconditions for an inductive proof are set up, and we can move forward with the proof.

We then need to prove the theorem T for all possible lengths of the list using induction. To accomplish this we call the `induct` tactic, which takes in the list and a inductive fact I. I must be a fact whose truth implies that T is true, and which can be proven inductively. The tactic comes with a baked in concept of an inductive header and inductive footer, which are equivalent to the top and bottom of our list. By letting I be that the bottom of the footer is below the top of the header we have an inductive fact that fits our conditions.

Cassius uses this list of tactics to build a group of Z3 queries about our page and then passes them through its internal Z3 engine. If all of those queries verify, then we can say that T is true for all possible configurations of the page and lengths of the list.

```
(component list (id list))
(component button (tag button))
(components todos (child (id list) *))

(havoc todos :text)
(havoc todos :w)
(assert todos (float-flow-skip ?))
(assert todos (and (> (height ?) 0) (non-negative-margins ?)))
(pre list (= (floats-tracked ?) 0))
(induct list ; the inductive fact
      (and (>= (bottom inductive-footer) (top inductive-header))
           (= (floats-tracked inductive-footer)
              (floats-tracked inductive-header))))
```

Figure 4.3. Above is the full proof for the T. This proof works by splitting the page into the list, button, and items that make up the list, and then proving that T is true about our list using `assert` to get Z3 to check preconditions,`pre` to create user established preconditions that are implied by those asserts, `havoc` to remove constraints on the proof, and `induct` to perform a proof by induction on the list itself.

CHAPTER 5

METHODS

5.1 Verifying Javascript

We cannot verify a dynamic web page without taking the page's JavaScript into account. JavaScript is infamous for its edge cases. Therefore creating a tool that can verify arbitrary JavaScript code is outside the scope of this project. Instead, we redefine JavaScript as a list of changes that can change a page's state, and then prove our fact is true for all possible states.

To start this process, we convert the page's JavaScript into a list of changes. We expand the capture script's IR to be able to represent a very small subset of JavaScript, which can be explained through the following Esprima AST nodes. First, there are two types of `VariableDeclaration` nodes the capture script recognizes: a `querySelector` node and a `createElement` node, each of which correlates with the JavaScript command of the same name. The capture script responds to these by adding the variable being declared to a local environment. Next, the capture script is able to understand the `AssignmentExpression` AST node, which happens any time the JavaScript sets a value to some element associated with a previously declared variable. The capture script IR havocs the value, which evaluates the proof regardless of the value, as will be further explained in Section 5.2. Finally, the capture script can understand the `appendChild` expression, and responds by adding the given child to the correct parent. All other lines of the JavaScript are havoced.

The above JavaScript AST nodes give us a way of understanding two different changes to the page that are necessary to verify our case study. First, when an element's attribute changes, that attribute is havoced. Second, an element on the page can gain new children.

After the capture script generates a list of possible changes to the page, these changes must be used to generate a new DOM tree. Luckily the capture script provides output that

simplifies this task. We iterate through the list and either add a variable to the environment tied to some element, havoc a value in a given element, or add an element as a child to a different element. Once we have performed all the changes in the list, we replace the old DOM tree of the page with the new tree that has been generated by applying these changes.

5.2 Havocing Information in Cassius

As briefly mentioned in Chapter 4, values in Cassius can be thought of less as actual values on a web page and more as constraints for a proof. To help explore this, let us look at the following example component.

```
([TEXT :x 48 :y 187 :w 83 :h 19 :text "value-10/3"])))
```

This describes one of the lines of text in our list. The `:x` and `:y` define the position of the text, the `:w` and `:h` define the height and width of the text, and the `:text` defines what the actual letters of the text are. Cassius translates this to a series of Z3 queries, which can be seen below.

```
(assert (! (= (box-x box2037) 48) :named box-x/2037))
(assert (! (= (box-y box2037) 187) :named box-y/2037))
(assert (! (= (box-width box2037) 83) :named box-width/2037))
(assert (! (= (box-height box2037) 19) :named box-height/2037))
```

In our case study we havoc the width of the text. To do this we use the `havoc` tactic to specify in the proof that the width should be removed, and then instead of the above set of Z3 queries, Cassius produces the following:

```
(assert (! (= (box-x box2037) 48) :named box-x/2037))
(assert (! (= (box-y box2037) 187) :named box-y/2037))
(assert (! (= (box-height box2037) 19) :named box-height/2037))
```

This means that the proof becomes true for all possible widths of the text, allowing our proof framework to correctly respond to the first change that can be caused by the JavaScript, where information changed by the script is havoced..The second, appending a child to the list, is handled by the inductive proof described in the next section

5.3 Induction in Cassius

Much of the actual work in this project revolves around proving a theorem T on a list by induction, which is difficult because inductive proofs introduce a high level of technical complexity into the system. This is because Cassius is built on top of Z3, an SMT solver which, due to the nature of SMT, is not capable of inductive logic. This means that we will only be able to use Cassius to prove the different cases of an inductive proof, and will have to do the actual logical leaps between those cases ourselves.

5.3.1 Preconditions of an Inductive Proof

Before we can perform a proof by induction we must check that the list we want to induct over meets the preconditions for induction. Specifically, we need to check that the elements of the list are similar enough to one another for the logic behind induction to be true.

There are two ways that boxes in our list can be meaningfully dissimilar. The first is whether a box has next and previous pointers to other boxes. While most of the children of the list do, the first child lacks a previous pointer and the last child lacks a next pointer. This means we exclude those boxes from our inductive proof. To still have a valid proof we prove that T applies to these boxes by making Cassius directly prove T on versions of the list that are 1, 2, and 3 elements long.

The second way this precondition can fail is if the attributes of different boxes are different. To check this we iterate through the list and check that all boxes contain the same attributes. We can also use the `havoc` tactic mentioned above to erase that some difference between elements has no real effect on the proof and remove it from the check. If the list passes this test we can safely perform a proof by induction on it.

5.3.2 Formal Description of Induction

Before going any further let us formally define what an inductive proof actually is, specifically in the context of our use case. For the theorem T, described in Chapter 4, to be proven inductively we must have an inductive fact $I(A, B)$ where I is a property of the list from an arbitrary box in the list A to a later arbitrary box B. To prove this implication we must first prove $I(A, B)$, and to start that we must prove P on a base case

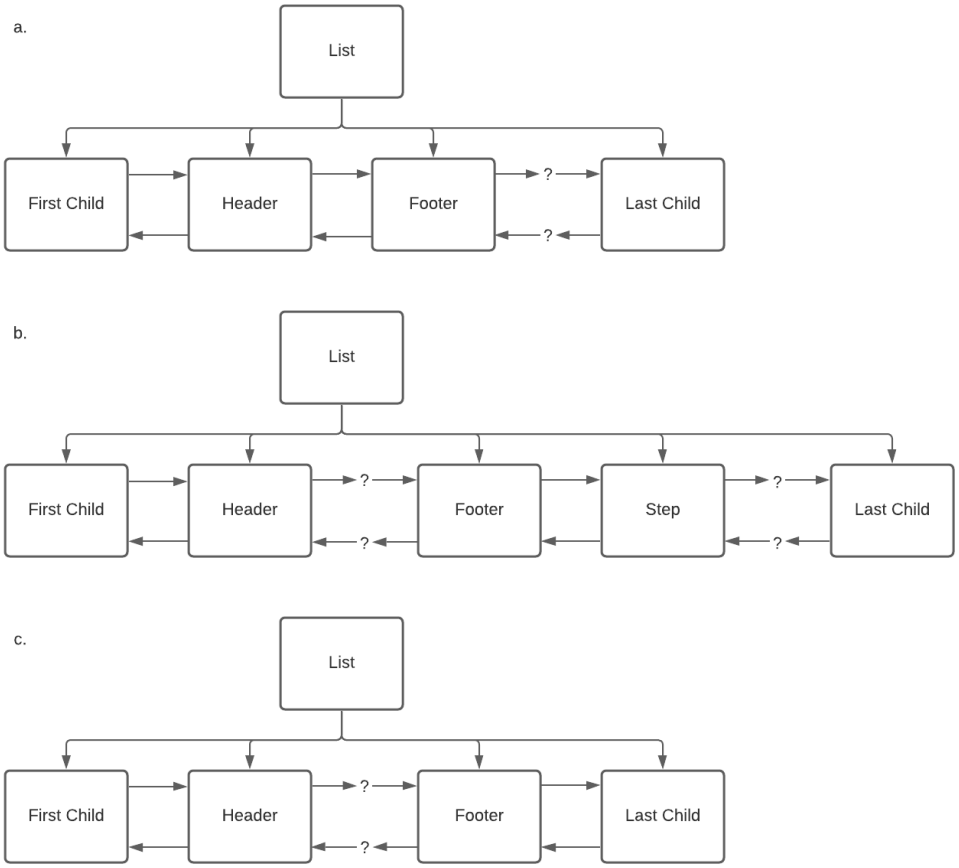

Figure 5.1. Above are the trees that represent what the list in our page looks like for the base case, inductive step, and theorem case of our inductive proof. In each case we leverage Cassius's ability to havoc information about what happens between two elements of the list. This lets us set up a base case (a.) that is the first three elements of the list, followed by an unknown number of different things in the list, followed by the final element. The inductive step (b.) takes this one step further, where the first three elements and the havoced space in between let us have a list of N elements, and tacking on the step node lets us have a list of N+1 elements, where the space after is havoced so that this works no matter where in the list the N and N+1 case end up being. Finally the theorem case (c.) defines a list of unknown length that we use to show that our inductive fact implies that our original theorem is true.

$BC = P(B_2, B_3)$. As mentioned above we avoid the first and last boxes in our proof, so our base case involves the second box B_2 and the third box B_3. If the base case is true then we next must prove the inductive step, $IC = I(B_2, B_N) \rightarrow I(B_2, B_{N+1})$. If Pre, BC, and I are all true then $P(A, B)$ is true, and from here $TC = I(A, B) \rightarrow T$ can be used to inductively prove T.

5.3.3 Base Case

The first step of the inductive proof described above is BC, so we start our verification there. The structure of this list can be seen in Figure 5.1a, where the ? signals the next and previous pointers for those boxes have been havoced. Thus, Cassius allows an unknown number of nodes between the third and fourth nodes, which allows us to think of this as looking at the first 3 elements of a list of unknown length. We then have Cassius verify T about this subsection of the list, thereby generating and proving BC, and allowing us to move forward with the inductive proof.

5.3.4 The Inductive Case

Next we prove IC. The structure of this list can be seen in Figure 5.1b, where once again the ? signals which next and last pointers have been havoced. This allows Cassius to see this as a list of unknown length where we are looking at two consecutive elements at an unknown point in the list. In other words, we are looking at the Nth and and N+1th element of the list.

Now that we have something of the correct shape we can verify I. To replicate the traditional inductive logic where one assumes the inductive fact is true for an N case and then uses this to prove its true on the N+1 case, we will feed the following implication into Z3: $I(B_2, B_N) \rightarrow I(B_2, B_{N+1})$. If Z3 verifies this, then we know that the I is true.

5.3.5 Proving the Theorem

Finally, we must prove $P(A, B) \rightarrow T$. To do this we will create one last copy of the list that looks like Figure 5.1c. We take this version of the list, which as with the previous cases has havoced specific next and previous pointers to create a list of some unknown length greater than four, and have Cassius apply $P(A, B) \rightarrow T$ to said list. If this implication verifies then we can combine that with proofs described above to show that T is true.

CHAPTER 6

RESULTS

We created a page and proof like those described in Chapter 4 and performed the proof as described throughout the paper. Cassius was able to show that T was true for all possible configurations of the page and lengths of the list, something we can be confident is not a false positive given the simplicity of the page and our proof.

There was an additional benefit to the `induct` and `havoc` tactics in that it vastly improved Cassius's performance on lists in static web pages. Previously the system directly verified a list and every single one of its children, something that took a long time on lists of substantial length. We took a snapshot of our page from Chapter 4 after pressing the button 50 times. We than verified T on this static snapshot of a single state of the page, once using neither the havoc or induct tactics, once using the havoc tactics, and once using both the induct and the havoc tactics. The timing results can be seen in Table 6.1. Overall, the new tactics lead to a $5\times$ speed up on the verification of the static page.

No Changes	2m54s
Havok	47s
Havok and Induct	39s

Table 6.1. In addition to allowing us to prove facts about dynamic web pages, the inductive proof scheme also lead to sizeable increases to performance on static pages. Above are the effects of `havoc` and `induct` on a static version of our page with a list of 50 elements.

CHAPTER 7

CONCLUSION

While previous literature has formally verified static pages, little work has been done on the topic of formally verifying dynamic pages. We have shown that such verification is possible by expanding upon static verification systems to effectively verify an example page. This shows that future work verifying dynamic pages in general is a viable area of research.

REFERENCES

[1] ARIYA HIDAYAT. ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/, year unknown.

[2] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, p. 337–340.

[3] HALLE, S., BERGERON, N., GUERIN, F., AND LE BRETON, G. Testing web applications through layout constraints. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), pp. 1–8.

[4] MANUEL CARRETERO. Mozilla Firefox Default CSS. https://hg.mozilla.org/mozilla-central/file/tip/layout/style/res/html.css, 4 2021.

[5] PANCHEKHA, P., ERNST, M. D., TATLOCK, Z., AND KAMIL, S. Modular verification of web page layout. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019).

[6] PANCHEKHA, P., GELLER, A. T., ERNST, M. D., TATLOCK, Z., AND KAMIL, S. Verifying that web pages have accessible layout. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, Association for Computing Machinery, p. 1–14.

[7] PANCHEKHA, P., AND TORLAK, E. Automated reasoning for web page layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2016), OOPSLA 2016, Association for Computing Machinery, p. 181–194.