unjQuerify
Migration of jQuery Snippets to Modern Vanilla JavaScript APIs

Dereck J. Bridie
Delft University of Technology
Delft, The Netherlands
D.J.Bridie@student.tudelft.nl

Shinsuke Matsumoto and Shinji Kusumoto
Graduate School of Information Science and Technology
Osaka University, Osaka, Japan
{shinsuke, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—jQuery is a JavaScript library which can be used by developers when creating webpages. It gained popularity among web developers for normalizing web APIs in a time when browser incompatibilities were more common. However, modern browser vendors have adopted API standards which diminish the need for such a library. When these modern API standards are used instead of jQuery, page size is lowered, causing an improvement in page performance. This paper introduces a tool called unjQuerify which can transform snippets that make use of jQuery's functionalities into equivalent code that uses modern web API standards. unjQuerify also aids developers by displaying relevant documentation and the steps taken to transform the code, increasing developer familiarity with modern APIs.**

***Index Terms*—jQuery, JavaScript, browser incompatibility, vanilla, abstract syntax tree**

I. Introduction

JavaScript programmers often make use of libraries that provide convenient functionality as a layer above existing browser built-in APIs. An example of such a library is jQuery[1], a popular JavaScript library that boasts 97.0% market share [8]. It was created in 2006 by John Resig [5] to simplify web APIs such as HTML document traversal and manipulation in a time when browser incompatibilitiy problems [4] were common [5]. The most well-known browser incompatibility problem concerns the instantiation of Ajax request objects. An example of how it was constructed so that the Ajax object could work across different browsers is illustrated as below.

```
var xhr;
if (window.XMLHttpRequest){
  // If Chrome, Firefox, IE7+, Opera, Safari
  xhr = new XMLHttpRequest();
}
// otherwise, use ActiveX for IE5.x and IE6
else if (window.ActiveXObject) {
  try {
    xhr = new ActiveXObject("MSXML2.XMLHTTP");
  } catch (e) {
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
  }
}
// Initialize request
xhr.open("GET", "http://example.com");
...
```

Listing 1. Instantiation of Ajax object without jQuery

[1]jQuery: https://jquery.com/

This verbose instantiation process can be encapsulated by jQuery. The `$` identifier is an entry point to use all jQuery functions.

```
$.ajax({
  type: "GET",
  url: "http://example.com",
  ...
});
```

Listing 2. Instantiation of Ajax object with jQuery

Over the years, browsers have evolved to take World Wide Web Consortium's standards into consideration and have become largely standards-compliant. A large majority of users make use of such a standards-compliant browser [9], meaning that the need for jQuery to bridge functional incompatibilities has been diminished. Despite this, more than 73.3% of websites [8] continue to make use of the library.

Using jQuery can come with undesirable costs. Unnecessary usage of jQuery can lead to performance penalties [10] due to transfer speeds and parsing. These penalties are often not mitigated by the usage of Content Delivery Networks due to fragmentation of network providers and jQuery versions [6]. Furthermore, the weight of jQuery also has an impact on the loading times of a website. The size of version 3.3.1 of jQuery is 35kb (minified and gzipped). This is 10% of the median transfer size of JavaScript and JSON data used in webpages [1]. When uncompressed, this rises to a size of 87 kB, which may be order of magnitudes larger than the code written by a developer. Aside from the costs of the data transfer, parsing and executing the code of jQuery also takes extra time, whereas modern vanilla JavaScript APIs are already present in the browser and therefore carry no extra costs. As the popularity of smartphones increase, we must consider the high performance impact [11] on constrained computing devices as well. Also, reliance on any framework can introduce technical debt [2], [7].

For these reasons, it can be useful for a developer to consider removing their reliance on jQuery in their webpages. However, given the popularity of jQuery, it is possible that a developer is unfamiliar with modern API standards. In addition, maintaining and repairing steady JavaScript resources may need to be decisive because of the difficulty of equivalence checking [3]. Developers should be assisted in this migration from code using jQuery to code without it.

In this paper, we introduce a tool called unjQuerify that can remove jQuery usages from snippets using AST transformations. Given a snippet of JavaScript that makes use of jQuery functionality, a translation is given that have near-equivalent functionality, but make use of modern vanilla JavaScript APIs instead of jQuery functionality. Furthermore, in the interest of developer education, documentation is given that is relevant to each transformation.

II. Motivating Example

To explore the feasibility of such transformations, an sample will be given of jQuery code:

```
$("#btn").click(function() {
  $("#result").addClass("important")
      .text("The button was clicked.");
  $("#btn").hide();
});
```

Listing 3. Sample jQuery snippet

This snippet defines a click event listener for the element `#btn` using the `click(fn)` function. When the button is clicked, the class `"important"` is added to the `#result` element using the `addClass(class)` function. Also, its inner-text is changed. Finally, the button is hidden using `hide()` function to prevent multiple clicking.

For these expressions, vanilla DOM APIs exist that match these functionalities almost precisely. Examples of such expressions are:

- `click(fn)`: Adds a function that should be applied when the selected element is clicked. In modern vanilla APIs, this has the same functionality as `Element.addEventListener("click", function)`.
- `addClass(class)`: Adds the class(es) given in the first parameter to the selected elements. Using DOM APIs, this functionality is related to `Element.classList.add(name)`.
- `hide()`: Hides a element. In modern vanilla APIs, this is similar to `Element.style.display = "none"`.

Equivalent code can be expressed as the following:

```
const button = document.getElementById("btn");
button.addEventListener("click", function() {
  const res = document.getElementById("result");
  res.classList.add("important");
  res.textContent = "The button was clicked.";
  button.style.display = "none";
});
```

Listing 4. Sample manual conversion

However, some jQuery features must be taken into account when converting such code, as not all expression trees can be easily mapped to vanilla code. One example of this is jQuery's method chaining, which allows methods calls to follow each other without needing new statements. An example of this is the expression `$("div").filter(".post").hide()`, which has the same functionality as `let x = $("div"); x = x.filter(".post"); x = x.hide();`.

Furthermore, not all jQuery features can be transformed to modern idiomatic vanilla JavaScript alone. An example of this is `fade`, which should be implemented as a CSS transition in webpages following best practices. These features will not be transformed by this tool; instead, migration documentation will be shown.

III. Tool Overview

unjQuerify is a command-line tool that makes use of static analysis to transform a given source file that uses jQuery to DOM APIs using a set of rules. An abstract syntax tree (AST) is built, and registered rules are used to transform expression nodes to new nodes that use DOM standards. The tool is also publicly available on https://www.unjquerify.com through a web interface.

A. Architectural design

Figure 1 depicts a high-level overview of unjQuerify. unjQuerify uses `babel`[2] to construct ASTs from JavaScript source snippets and apply AST transformations.

The tool provides tree transformations that convert jQuery expressions to vanilla DOM expressions. Each transformation is annotated with documentation from the jQuery API guidebook, documentation from Mozilla's Web Docs[3], and information from Can I Use? browser compatibility information[4]. Then, `babel` is used again to generate JavaScript source code from the modified AST.

In the web presentation of the tool, each transformation is displayed separately. For each transformation that was applied, the documentation information is displayed along with the chunks that were affected by a transformation.

B. Implementation

The following sections detail the steps taken by unjQuerify after the input source code is converted to an AST.

1) Finding jQuery usages: jQuery defines two identical entrypoints into the library: `jQuery` and `$`. From there, this entry point can be used as a function (e.g. `$("#btn")`) or as a member property of the identifier (e.g. `$.isArray(id)`). Where applicable, function call chains are then dismantled into links, which are then seperately processed by unjQuerify's plugins.

2) Plugins: An unjQuerify plugin defines a node search strategy and a node replacement strategy. In the interest of expandability, to eventually be able to transform the entirety of jQuery's functionality, plugins can be defined and added to unjQuerify. These plugins are used when visiting each node in the input tree, and the given replacement is executed when a match is found.

3) Source generation and display: Each mutation to the source tree made by the tool is tracked. This information is used in the user interface as shown in Section III-C. The output code is generated by converting the mutated AST back into JavaScript code.

[2]https://babeljs.io/
[3]https://developer.mozilla.org/en-US/
[4]https://caniuse.com/

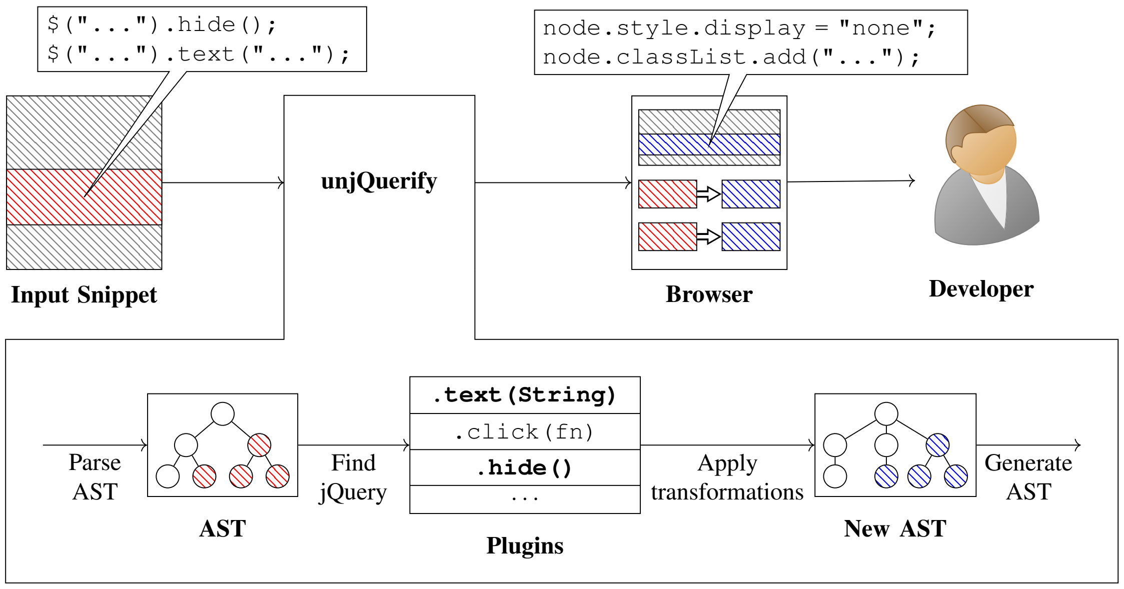


Fig. 1. unjQuerify architecture

C. *User presentation*

Another component of unjQuerify is its web interface. An actual screenshot of web interface is shown in Figure 2. The snippet on the left side is unjQuerify input that uses jQuery functions. The snippet on the right side shows the transformed (i.e., vanilla) snippet. Input can be supplied from a variety of sources, as well as from samples. These samples aim to ease a developer into the usage of unjQuerify before use with their own code. The modified code is displayed at the top. Under it, each transformation step taken to arrive at this result is shown. For each step, before and after snippets are shown, and relevant documentation can be found that elaborates upon the vanilla APIs which were used. Figure 3 shows an example of such documentation.

D. *Challenges in transformations*

1) Method Call Chains: As mentioned before, jQuery methods can be chained for ease of use. However, since vanilla APIs do not support such a style of calling functions, chained methods must be converted to support such expressions. When unchaining, intermediate results are assigned to temporary variables. However, this brings a set of challenges:

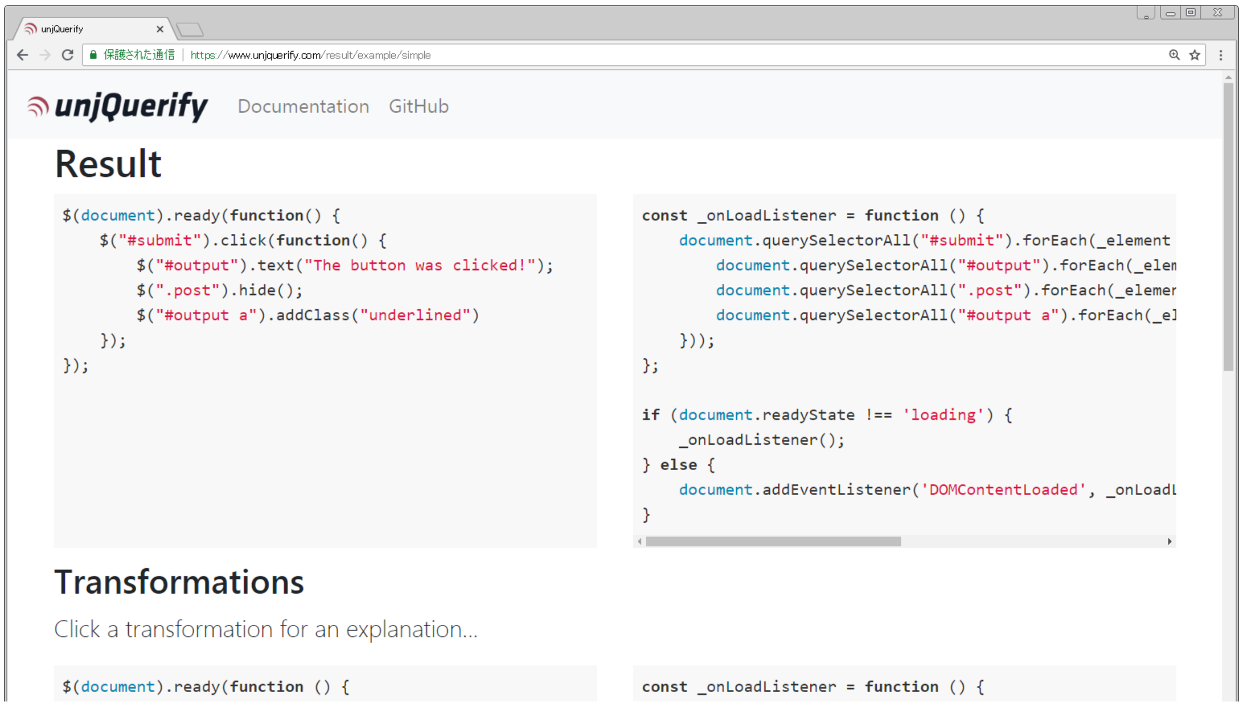
- Chains do not always operate upon the same collection. An example of this is `filter(String)`, which can reduce the number of matched elements in a collections. This means the new reduced collection must be assigned to a new variable in order to continue the chain on the correct set of elements.
- If a chain is on the right-side of a variable assignment, the last temporary variable should match the assignment of the original assignment.
- Unnecessary temporary variables should be avoided. An example of this is `hide()`, which can be used in a chain and does not cause a mutation of the original object. In this case, creating a new temporary assignment creates unnecessary clutter for a user to read.

unjQuerify adheres to the above rules when unchaining expressions.

2) Collections: Generally, jQuery methods that mutate document state operate on all elements in a selection. However, there are other methods that do not: an example is `is(String)`, which checks each element in a collection to see if at least one element matches a selector. Conversely, `css(String)` retrieves the value of an element collection's style property, but only using its first element. Each plugin in unjQuerify is aware of which elements a function operates upon.

3) Parameter Ambiguity: Some jQuery functions exist that accept multiple types within a given argument. An example of this is `$.each(Object|Array, callbackFn)`, which loops over the entries of an array when given an array, and loops over the keys and values in an object when given an object. An idiomatic transformation does not exist that accepts both objects and arrays in vanilla APIs. Therefore, when the type of this parameter cannot trivially be statically inferred to distinguish between these cases, a satisfactory idiomatic transformation cannot be given. In these cases, the tool must give a longer, non-idiomatic transformation.

4) Non-similarity: There are methods in jQuery that do not have a direct one-to-one mapping with vanilla functions. For example, `$element.parentsUntil(selector)` finds the ancestors of each element in the current set of matched elements up until the element matched in `selector`. Though this is one function call in jQuery, the equivalent function in vanilla APIs requires a helper function which is comparatively very verbose.


unjQuerify
保護された通信 | https://www.unjquerify.com/result/example/simple
unjQuerify Documentation GitHub

Result

```
$(document).ready(function() {
    $("#submit").click(function() {
        $("#output").text("The button was clicked!");
        $(".post").hide();
        $("#output a").addClass("underlined")
    });
});
```

```
const _onLoadListener = function () {
    document.querySelectorAll("#submit").forEach(_element
        document.querySelectorAll("#output").forEach(_elem
        document.querySelectorAll(".post").forEach(_elemer
        document.querySelectorAll("#output a").forEach(_el
    }));
};

if (document.readyState !== 'loading') {
    _onLoadListener();
} else {
    document.addEventListener('DOMContentLoaded', _onLoadl
}
```

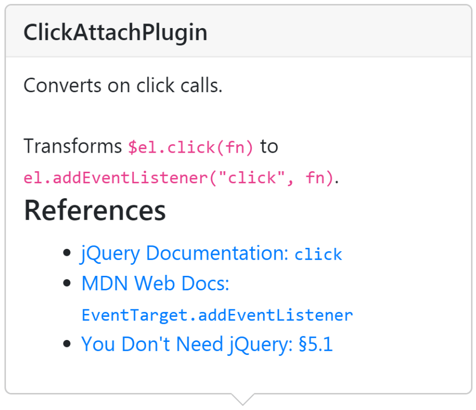
Transformations

Click a transformation for an explanation...

```
$(document).ready(function () {
```

```
const _onLoadListener = function () {
```

Fig. 2. Screenshot of unjQuerify web interface.



ClickAttachPlugin

Converts on click calls.

Transforms `$el.click(fn)` to `el.addEventListener("click", fn)`.

References

- jQuery Documentation: `click`
- MDN Web Docs: `EventTarget.addEventListener`
- You Don't Need jQuery: §5.1

Fig. 3. Example of documentation given for a `.click()` transformation.

IV. Preliminary Evaluation

unjQuerify's transformations should not introduce unsuspected behavioral changes, preventing code breakage after transformation. A transformation that introduces a behavioral change can cause bugs which do not become apparent from visually inspecting the transformed code. Therefore, it is important to verify that unjQuerify's transformations result in code that is functionally equivalent to the input code. Work has been started to attempt to verify this property, described in the following sections.

A. *jQuery unit tests*

jQuery uses a test-suite to verify the functionality of its features. We modify these tests to verify the correctness of the transformation by applying selective mutations to the tested jQuery code. An example of a unit test in jQuery can be found in code sample 5.

```
1 $("#nothidden").css("margin-top", "-10px");
2 var val = $("#nothidden").css("margin-top");
3 assert.equal(val, "-10px");
```

Listing 5. Sample jQuery test

When a transformation is known for the mutator of a CSS property `$element.css(property, value)`, its accessor `$element.css(property)` can be used to confirm that the mutator works as expected. unjQuerify's correctness tests use this property by, in the example, running one test with line 1 transformed but not line 2, and another test with line 2 transformed but not line 1. In this manner, both the accessor and the mutator have been verified.

This manner of transforming tests can be automated. Such a modified test has been used to test the functionality of the mutators and accessors of CSS properties, and should be extended to other functionalities in the future.

B. *Application to small code sample set*

For transformations that have been implemented, snippets from QuackIt jQuery Examples[5] have been collected and have informally been used to verify that these samples behave the same before and after transformations. These samples have been grouped into 12 groups, each covering an aspect of jQuery's functionality. For 8 of these groups, unjQuerify provides transformation plugins. Within these 8 groups, 32

[5]https://www.quackit.com/jquery/examples/

of the 49 given examples appear to work correctly after unjQuerify's transformations.

Using these tests, it has become clear that there are still some difficult constructions that must be addressed. For example, jQuery's `.click(fn)` function also supplies the element which was clicked in the implicit `this` context, whereas the vanilla JavaScript equivalent does not. This causes a runtime error in the output code. Future work should consider these cases.

V. Future Work

As this tool is still a work-in-progress, unjQuerify can still be improved. The following subsections elaborate upon several ways in which the tool can be extended.

A. Introduction of types

unjQuerify does not attempt to statically infer the types of identifiers. Therefore, each identifier which uses known jQuery methods assumed to be a reference to a jQuery collection. Code sample 6 illustrates a trival example of a heuristic possibly being used incorrectly.

```
const a = { hide() { console.log("Hidden"); }};
const b = $(".post");
function hide(element) {
  element.hide();
}
```

Listing 6. Example of unverifiable heuristic use

Without type information, it is not known if `element` used in the function parameter is a jQuery object or some other kind of object. In the given example, unjQuerify will incorrectly apply a transformation which assumes that `element` is a jQuery object that refers to HTML elements, resulting in a runtime error.

Projects exist that aim to improve tooling on JavaScript projects by introducing types, such as Microsoft's TypeScript[6] and Facebook's Flow[7]. Using type annotations would allow unjQuerify to be able to distinguish between these cases and apply transformations only when the type is correct.

B. Improve evaluation

In order to check the applicability and validity of unjQuerify for all jQuery snippets, using longer snippets could be valuable. For this, manual verification of small snippets and mutated jQuery unit tests have been used. However, this evaluation could be expanded upon by using large software repositories that are dependent upon jQuery and manually verifying the results of unjQuerify's transformations.

One dataset that could be used is NPM's list of jQuery's dependents[8]. NPM is a software package manager that can show which projects are dependent upon jQuery. Then, this dependent code can be transformed with unjQuerify, then manually verified or automatically verified using the project's own tests.

[6]https://www.typescriptlang.org/
[7]https://flow.org/
[8]https://www.npmjs.com/browse/depended/jquery

C. Increase compatibility

The compatibility of the given transformations can vary from browser to browser. Developers may want to target other browsers than only the newest set of ”evergreen” browsers, which boast the highest compatibility with browser standards. In this case, some of the given transformations might not be compliant with older browsers. It could be useful for unjQuerify to be able to suggest transformations that are compatible with such browsers as well.

Furthermore, unjQuerify has only been used with a single version of jQuery (v3.3.1). Since older versions of jQuery can be used within a production environment, the effects of the version changes should be evaluated and adjustments should be made to unjQuerify to account for this.

VI. Conclusion

In this paper, a tool was presented to offer suggestions to transform jQuery code snippets into modern vanilla JavaScript code based on AST transformations. By using this tool, developers can migrate their code so that it does not depend on this library. Furthermore, developers can use the references provided to increase their familiarity with modern documentation resources. Testing has been done with jQuery's unit tests and small sample code sets. In the future, more progress should be made to increase the accuracy and applicability of the transformations provided by unjQuerify.

Acknowledgements

This work was supported by JSPS/MEXT KAKENHI Grant Number 16H02908 and 18H03222. It could not have been possible without JASSO scholarship funding and Osaka University's FrontierLab program.

References

[1] HTTP Archive. Page weight. https://httparchive.org/reports/page-weight#bytesJs. Accessed: 20 June 2018.

[2] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, Nov 2012.

[3] Nuno P. Lopes and José Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer*, 18(4):359–374, Aug 2016.

[4] Ali Mesbah and Mukul R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 561–570, New York, NY, USA, 2011. ACM.

[5] Charles Severance. John Resig: building JQuery. *Computer*, 48(5):7–8, 2015.

[6] Steve Souders. HTTP archive: jQuery. http://www.stevesouders.com/blog/2013/03/18/http-archive-jquery/. Accessed: 20 June 2018.

[7] Edith Tom, AybüKe Aurum, and Richard Vidgen. An exploration of technical debt. *J. Syst. Softw.*, 86(6):1498–1516, June 2013.

[8] Usage statistics and market share of JavaScript libraries for websites, june 2018. https://w3techs.com/technologies/overview/javascript_library/all. Accessed: 20 June 2018.

[9] Can I Use? Browser usage table. https://caniuse.com/usage-table. Accessed: 20 June 2018.

[10] TJ VanToll. Is jQuery too big for mobile? https://modernweb.com/is-jquery-too-big-for-mobile/. Accessed: 20 June 2018.

[11] Bo Zhao, Byung Chul Tak, and Guohong Cao. Reducing the delay and power consumption of web browsing on smartphones in 3g networks. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 413–422. IEEE, 2011.