Refactoring towards the Good Parts of JavaScript

Asger Feldthaus ∗
Aarhus University
asf@cs.au.dk

Todd Millstein †
University of California,
Los Angeles
todd@cs.ucla.edu

Anders Møller ∗
Aarhus University
amoeller@cs.au.dk

Max Schäfer
University of Oxford
max.schaefer@cs.ox.ac.uk

Frank Tip
IBM Research
ftip@us.ibm.com

Abstract

JavaScript is one of the most widely used programming languages of the present day. While its flexibility is treasured by proponents, its lack of language support for encapsulation is an obstacle to writing maintainable programs. We propose refactorings for improving modularity, and discuss challenges arising in their implementation.

Categories and Subject Descriptors D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering

General Terms Languages

1. Introduction

Recent years have witnessed a resurgence of dynamic programming languages, such as JavaScript and PHP, which eschew the focus on static typing and modularity constructs of languages like Java, promising less development overhead and powerful dynamic techniques. The domain of web programming, in particular, is dominated by such languages.

While these features help with rapidly writing new code, they often become a liability once programs grow beyond a certain size, and can be an obstacle to reusability and integration. Modern textbooks on JavaScript hence put particular emphasis on programming patterns and idioms that can be used to modularize and structure code, advocating the use of what a popular textbook terms the “good parts” of the language [1]. Additionally, the new ECMAScript 5 standard for JavaScript provides for a so-called strict mode that disallows the use of some problematic language constructs, and restricts the use of others [2].

The experience of the ill-fated ECMAScript 4 effort has shown that major changes or additions to the language are unlikely to happen in the near future. Instead, programmers have to be persuaded to migrate their programs to safe sub-languages or employ recommended idioms and patterns to make up for missing language features where possible.

However, doing this by hand can be a difficult task that may involve changing many different parts of the program and reasoning carefully about aliasing patterns and side effects. Hence programmers would benefit from having tool support for performing such refactorings.

A refactoring tool for a language like JavaScript can assist with enforcing best practices, eliminating uses of problematic language features, and enabling programs to make use of higher-level frameworks and domain specific languages. Refactoring can thus play an active role in mitigating deficiencies of the refactored language and raising the level of abstraction at which a programmer can work.

2. Modularization Refactorings

This section briefly introduces two modularization refactorings for JavaScript that refactor towards patterns recommended in the literature [1]. More detailed specifications of these refactorings can be found in the companion paper [3].

The ENCAPSULATE PROPERTY refactoring encapsulates a property of an object, rerouting all accesses to the property through accessor methods.

An example application of this refactoring is shown in Fig. 1. The original program, shown on the left, defines a constructor function Rect. When invoked on a newly created object, this function initializes properties width and height of the object to the values passed in parameters w and h. On line 5, the function’s prototype property is assigned an object literal containing a method area that will be available on all objects constructed using Rect. Finally, a new Rect
3. Challenges

Carefully specifying and implementing refactorings is a challenging task even for statically typed languages like Java, and it is considerably more difficult for JavaScript.

Consider, for instance, the **ENCAPSULATE PROPERTY** refactoring. After transforming the given property into a local variable, we have to determine which property accesses need updating. This is tricky, since properties are not declared in JavaScript but are created upon first write. Clearly, a form of static analysis is needed to (conservatively) determine whether two property accesses can potentially refer to the same property at runtime.

Static analysis of JavaScript is a fairly new, but quite active field of research. In order to prevent prematurely committing ourselves to one particular analysis technique, we have instead defined an abstract interface between the analysis engine and the refactoring specifications, consisting of a set of queries that the analysis has to implement.

We have specified and implemented the refactorings **RENAME**, **ENCAPSULATE PROPERTY** and **EXTRACT MODULE** in terms of this interface, using a custom pointer analysis to implement the queries. The implementation, including an integration into Eclipse as a plugin, is available online at www.brics.dk/jsrefactor.

The dynamic features of JavaScript are particularly difficult to handle. Our specifications conservatively handle reflective features such as `for-in` loops and computed property names: renaming a property that could be accessed reflectively is forbidden. However, `eval` and equivalent constructs are not handled at all; instead, we emit a warning about possible unsoundness if a use of `eval` is detected.

4. Conclusions

We have argued for the use of tool-supported refactoring for JavaScript to enable programmers to restructure their programs and make use of patterns and best practices. We have described two examples of such refactorings that can be used to improve program modularity, and we have discussed some of the challenges arising in their implementation.

We believe our approach points the way to a powerful new usage of automated refactorings as a proxy for language extension. While JavaScript lacks some commonly desired modularity and encapsulation mechanisms, with appropriate refactoring support developers can effectively add these mechanisms to the language and reliably use them in programs. In a sense, refactoring towards the “good parts” contributes to building a better JavaScript language.

References


---

```javascript
function Rect(w, h) {
    this.width = w;
    this.height = h;
}
Rect.prototype = {
    area: function() {
        return this.width * this.height;
    },
    *this.getWidth = function() {
        return width;
    },
}
rect = new Rect(23, 42);
alert(rect.width);
```

Figure 1. Using **ENCAPSULATE PROPERTY** to turn property `width` into a local variable of the constructor `Rect`.

Object is constructed on line 11, and the value of its `width` property displayed using an alert dialog.

Since JavaScript has no access control for properties, the properties `width` and `height` of a `Rect` object can be freely read, written or even deleted by any code that has a reference to it. The **ENCAPSULATE PROPERTY** refactoring makes it possible to restrict access to a property by a well-known technique shown in the refactored program on the right, where changes have been highlighted.

The former property `width` has now been turned into a local variable of the constructor function and is thus no longer accessible from outside `Rect`. To compensate, the refactoring adds an accessor method `getWidth` that returns the value of `width`. Former read accesses to the property `width` on lines 23 and 28 have to be adjusted to use the method instead. If there had been a write access, the refactoring would additionally have introduced a setter method `setWidth`.

Thus, while the transformation from a property to a local variable looks innocuous enough, it necessitates global adjustments that are non-trivial to perform by hand. In addition, the refactoring will have to check several preconditions to ensure that this transformation is behavior-preserving. For instance, there cannot already be a method `getWidth`, and the program cannot use the reflective features of JavaScript to access properties of `Rect` objects [3].

This is a typical example how a refactoring (**ENCAPSULATE PROPERTY**) emulates a missing language feature (access control) by making a workaround easily available.

Another, similar example is the refactoring **EXTRACT MODULE**: JavaScript provides no modularity constructs, which often leads to a proliferation of global definitions, making programs brittle and hard to integrate. Utilizing a well-known module pattern [1], this refactoring packs global definitions into a record with a shared scope provided by a closure, and adjusts their uses all over the program. Again, a refactoring makes up for a missing language feature by facilitating the use of a workaround.