Tool-supported Refactoring for 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

Refactoring is a popular technique for improving the structure of existing programs while maintaining their behavior. For statically typed programming languages such as Java, a wide variety of refactorings have been described, and tool support for performing refactorings and ensuring their correctness is widely available in modern IDEs. For the JavaScript programming language, however, existing refactoring tools are less mature and often unable to ensure that program behavior is preserved. Refactoring algorithms that have been developed for statically typed languages are not applicable to JavaScript because of its dynamic nature.

We propose a framework for specifying and implementing JavaScript refactorings based on pointer analysis. We describe novel refactorings motivated by best practice recommendations for JavaScript programming, and demonstrate how they can be described concisely in terms of queries provided by our framework. Experiments performed with a prototype implementation on a suite of existing applications show that our approach is well-suited for developing practical refactoring tools for JavaScript.

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

General Terms Languages

* Supported by the Danish Research Council for Technology and Production, grant no. 09-064210.

† Supported by the US National Science Foundation, award CCF-0545850.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

Refactoring is the process of improving the structure of software by applying behavior-preserving program transformations [9], and has become an integral part of current software development methodologies [4]. These program transformations, themselves called refactorings, are typically identified by a name, such as RENAME FIELD, and characterized by a set of preconditions under which they are applicable and a set of algorithmic steps for transforming the program's source code. Checking these preconditions and applying the transformations manually is tedious and error-prone, so interest in automated tool support for refactorings has been growing. Currently, popular IDEs such as Eclipse,[1] VisualStudio,[2] and IntelliJ IDEA[3] provide automated support for many common refactorings on various programming languages. In addition, there is much recent research literature on soundly performing a variety of refactorings (see Section 7 for an overview).

However, most research on refactoring has focused on statically typed languages, such as Java, for which expressing the preconditions and source code transformations can take advantage of static type information and name resolution. Refactoring for dynamic languages such as JavaScript is complicated because identifiers are resolved at runtime. Of the few previous approaches to refactoring for dynamically typed languages, the most well-developed one can be found in the Smalltalk Refactoring Browser [24], which relies on a combination of runtime instrumentation and the existence of a test suite to ensure that behavior is preserved. By contrast, we aim for a sound technique that does not require comprehensive test suites.

In this paper, we present a framework for refactoring programs written in JavaScript, a dynamically typed scripting

[1] `http://www.eclipse.org/`

[2] `http://www.microsoft.com/visualstudio/`

[3] `http://www.jetbrains.com/idea/`

language that has become the lingua franca of web browsers. To understand why implementing even simple refactorings in JavaScript is more challenging than implementing analogous refactorings for statically typed languages such as Java, consider the RENAME FIELD refactoring in Java. A key requirement when renaming field `f` of class `C` to `g` is to identify all references to that field so they can be renamed consistently. Renaming all references to a field is easy for Java programs since static type information is available. For example, an expression of the form `e.f` where the static type of `e` is `C` definitely refers to the renamed field. In contrast, the corresponding task for a RENAME PROPERTY refactoring in JavaScript is in general impossible to solve exactly by static means. While fields in Java are statically declared within class definitions, properties in JavaScript are only associated with dynamically created objects and are themselves dynamically created upon first write. Further complications arise from other dynamic features of JavaScript, such as the ability to dynamically delete properties, change the prototype hierarchy, or reference a property by specifying its name as a dynamically computed string.

We describe a methodology for implementing automated refactorings on a nearly complete subset of the ECMAScript 5 language [7], the chief omission being dynamically generated code (i.e., `eval`). Our approach relies on static pointer analysis for JavaScript to define a set of general analysis queries. We have used this methodology to implement both well-known traditional refactorings, such as renaming, and novel JavaScript-specific refactorings that target desirable programming idioms advocated by influential practitioners [5].

In the process, we have devised various techniques to handle JavaScript's highly dynamic features and lack of static typing. For example, while naively over- or under-approximating the set of expressions `e.f` that must be modified when a property `f` is renamed (e.g., using a conventional must- or may-point-to analysis) would be unsound, we describe an algorithm that over-approximates this set in a safe manner. We also ensure, through preconditions that can be expressed in terms of the analysis queries, that behavior is preserved in the presence of complex JavaScript features such as reflective `for-in` loops, first-class functions, and prototype-based inheritance. In cases where we cannot guarantee behavior preservation, refactorings are prevented from being applied.

We have specified and implemented three refactorings using our approach: RENAME (which is a generalization of the previously mentioned RENAME PROPERTY), ENCAPSULATE PROPERTY, and EXTRACT MODULE. We have evaluated the quality of our implementations by applying these refactorings systematically to a set of 50 benchmark programs, measuring how often refactorings are applied successfully and analyzing causes for rejection. Our results show that most refactorings are performed successfully and rejections are generally justified by a real danger of unsoundness. This demonstrates that our approach is a viable basis for implementing refactoring tools for JavaScript.

In summary, the major contributions of this paper are as follows:

- We present a **framework for specifying and implementing JavaScript refactorings**, based on a set of analysis queries on top of a pointer analysis.
- We give **concise, detailed specifications of JavaScript-specific refactorings** expressed using the framework. To the best of our knowledge, we are the first to give such specifications in the context of JavaScript.
- We **experimentally validate** our approach by exercising a prototype implementation of the framework and the refactorings on a set of JavaScript benchmarks. We demonstrate that the preconditions of our specifications are not overly conservative, and that a relatively simple pointer analysis appears to suffice in practice for many programs ranging in size from 300 to 1700 lines of code.

The remainder of this paper is organized as follows. Section 2 introduces a motivating example to illustrate the challenges that arise in defining several refactorings for JavaScript. Section 3 presents a framework of analysis queries based on pointer analysis. Section 4 shows how the three refactorings under consideration are expressed using this framework. Details of the implementation are described in Section 5, while Section 6 gives an evaluation of our refactorings on a set of JavaScript benchmarks. Related work is discussed in Section 7. Finally, conclusions are presented in Section 8.

2. Motivating Examples

Figure 1 shows a small JavaScript program that we will use to illustrate some of the challenges of refactoring JavaScript programs. Part (a) of the figure shows a library that defines two shapes: circles and rectangles. Part (b) shows a client application that uses this library to draw a number of such shapes of randomly chosen sizes at random coordinates in the browser. We will first explain some key details of this program, and then discuss some of the issues raised by applying the RENAME, ENCAPSULATE PROPERTY, and EXTRACT MODULE refactorings.

2.1 A JavaScript Example Program

As a prototype-based language, JavaScript does not have built-in support for classes. Instead, they are commonly simulated using *constructor functions*. In the example of Figure 1, two constructor functions are provided: `Circle` (lines 1–11) and `Rectangle` (lines 13–24). These enable the programmer to create circle and rectangle objects using the `new` operator (e.g., line 41). Constructor functions typically contain statements to initialize a number of object properties, which are not explicitly declared but created upon the

```
1  function Circle(x, y, r, c) {
2    this.x = x;
3    this.y = y;
4    this.radius = r;
5    this.color = c;
6    this.drawShape = function (gr) {
7      gr.fillCircle(new jsColor(this.color),
8                    new jsPoint(this.x,this.y),
9                    this.radius);
10   };
11 }
12
13 function Rectangle(x, y, w, h, c) {
14   this.x = x;
15   this.y = y;
16   this.width = w;
17   this.height = h;
18   this.color = c;
19   this.drawShape = function (gr) {
20     gr.fillRectangle(new jsColor(this.color),
21                      new jsPoint(this.x,this.y),
22                      this.width, this.height);
23   };
24 }
25 Rectangle.prototype.getArea = function() {
26   return this.width * this.height;
27 };
```

(a)

```
28 function r(n) { return Math.round(Math.random() * n); }
29
30 function drawAll(sh) {
31   var gr =
32     new jsGraphics(document.getElementById("canvas"));
33   sh.map( function(s) { s.drawShape(gr); });
34 }
35
36 var shapes = [];
37 for (var i = 0; i < 500; i++) {
38   var o = new jsColor().rgbToHex(r(255),r(255),r(255));
39   switch(r(2)){
40     case 0:
41       shapes[i] = new Circle(r(500),r(500),r(50),o);
42       break;
43     case 1:
44       shapes[i] = new Rectangle(r(500),r(500),r(50),r(50),o);
45       alert(shapes[i].getArea());
46       break;
47   }
48 }
49 drawAll(shapes);
```

(b)

Figure 1. Shapes example. Part (a) shows a small library that defines several types of shapes. Part (b) shows a small client application that uses the library to draw shapes in the browser.

first write. For example, the constructor for `Circle` creates and initializes properties `x`, `y`, `radius`, and `color` (lines 2–5) by assigning them values that are passed in as parameters to the function, and similar for `Rectangle`.

Both also create properties `drawShape` on line 6 and line 19 that contain functions to display the appropriate geometric shape.[4] These functions can refer to their receiver object using `this` expressions, and thus act like methods.

Function `Rectangle` shows another way of emulating methods that makes use of JavaScript's prototype-based nature. Functions like `Circle` and `Rectangle` are themselves objects, and hence can have properties. In particular, every function object has a `prototype` property that is implicitly initialized to an empty object. On line 25 we create a property `getArea` in this object by assigning it a function that computes the area of a rectangle. Every object created by invoking `new Rectangle(...)` has an *internal prototype* property, which references the object stored in `Rectangle.prototype`. When a property x is looked up on this object, but the object does not itself define property x, the internal prototype is searched for x instead.

Thus, every rectangle has both a `getArea` and a `drawShape` property, the latter defined in the object itself, the former defined in its internal prototype. But while every rectangle has its own copy of `drawShape` (created on line 19), there is only one copy of `getArea`, which is shared by all rectangles.

[4] The functions are implemented using `jsDraw2D`, a graphics library for JavaScript, which is available from `http://jsdraw2d.jsfiction.com/`.

Function `r` (line 28) returns a random value between `0` and its argument `n`. Function `drawAll` (lines 30–34) takes as an argument an array `shapes`, and on line 33 uses a closure and the `map` function to invoke `drawShape` on each element of the array. Lines 36–49 contain a sequence of statements that are executed when the page containing the script is loaded. This code creates an array of 500 randomly colored shapes of various kinds, displaying the area of every rectangle upon creation on line 45, and then invokes `drawAll` to draw these shapes in the browser.

Note that in the invocation `shapes[i].getArea()` on line 45, the function to be invoked is found on the internal prototype object of `shapes[i]`, but its receiver object (i.e., the value of `this`) is `shapes[i]` itself, not the prototype object. This ensures, for instance, that the property access `this.width` in line 26 refers to the property defined on line 16.

We will now discuss the issues that arise when three refactorings—RENAME, ENCAPSULATE PROPERTY, and EXTRACT MODULE—are applied to the example program of Figure 1.

2.2 RENAME

We begin by considering some applications of the RENAME refactoring to the example program of Figure 1. In JavaScript, there are no property declarations. Although it is natural to think of the assignment to `this.x` in `Circle` as a declaration, it is just a write access to a property `x` that is created on the fly since it does not exist yet. The absence of declarations and static typing information makes refac-

toring more difficult because it is necessary to determine all property expressions in a program that may refer to the same property and rename them consistently. We consider a few examples:

- The property expression `this.x` on line 2 in `Circle` can be renamed to `xCoord`. This requires updating the property expression `this.x` on line 8 to `this.xCoord` as well. However, there is *no* need to rename the property expression `this.x` on line 14, because the properties accessed on lines 8 and 14 must reside in different objects. If we nevertheless do decide to rename `this.x` on line 14 to `this.xCoord` as well, then the subsequent property expression on line 21 must also be changed to `this.xCoord`.
- Refactoring the property expression `this.drawShape` on line 6 in `Circle` to `this.draw` requires that the property expression `this.drawShape` on line 19 in `Rectangle` is refactored to `this.draw` as well: the receiver `s` in the expression `s.drawShape(gr)` on line 33 can be bound to a `Circle` or a `Rectangle` object, and therefore the methods have to be renamed consistently. Note that `Circle` and `Rectangle` are completely unrelated; in particular there is no prototype relationship.

As these examples illustrate, the key correctness requirement for renaming is *name binding preservation*—each use of a property in the refactored program should refer to the same property as in the original program. Name binding preservation is also a natural correctness condition for other refactorings, as we describe below. Schäfer et al. [26] used this condition successfully to provide sound automated refactorings for Java, including renaming. Unfortunately, their techniques rely on explicit declarations and static scoping, so they are not directly applicable to JavaScript.

A natural approach is to use a static pointer analysis to approximate name binding information. However, a naive use of pointer analysis would be unsound. For example, consider the renaming of `this.drawShape` on line 6 in `Circle` described above. Renaming only expressions that *must point to* the same property as the one referenced on line 6 is insufficient. A sound must-point-to analysis could indicate that there is no other access of `drawShape` that must definitely point to the same property, therefore requiring nothing else to be renamed. On the other hand, renaming only expressions that *may point to* the property referenced on line 6 is also insufficient. For example, a sound may-point-to analysis could exclude the property expression `this.drawShape` on line 19 in `Rectangle` since it definitely accesses a distinct property from that of `Circle`. However, that expression must in fact be renamed to preserve program behavior, as we saw above. We define a notion of *relatedness* in Section 3 based on may-points-to information, which captures the set of property expressions in a program that are affected by renaming a particular expression a.

```
50 function dble(c) {
51   var nc = new Circle();
52   for (var a in c) {
53     nc[a] = (a != "radius") ? c[a] : c[a]*2;
54   }
55   return nc;
56 }
57
58 function r(n) { return Math.round(Math.random() * n); }
59
60 function drawAll(sh) {
61   var gr =
62     new jsGraphics(document.getElementById("canvas"));
63   sh.map( function(s) { s.drawShape(gr); });
64 }
65
66 var shapes = [];
67 for (var i = 0; i < 500; i++){
68   var o = new jsColor().rgbToHex(r(255),r(255),r(255));
69   switch(r(2)) {
70     case 0:
71       shapes[i] =
72         dble(new Circle(r(500),r(500),r(50),o));
73       break;
74     case 1:
75       shapes[i] =
76         new Rectangle(r(500),r(500),r(50),r(50),o);
77       alert(shapes[i].getArea());
78       break;
79   }
80 }
81 drawAll(shapes);
```

Figure 2. Modified client application, with `dble` function added.

We now consider a minor variation on the client application where a function `dble` has been added (lines 50–56) as shown in Figure 2. The client application is the same as in Figure 1(b) except that `dble` is used to double the radius of circles created in line 72.

The function `dble` takes an argument `c`, which is assumed to be a `Circle` object, and returns a new `Circle` object at the same coordinates but with a doubled radius. This function illustrates several interesting features of JavaScript. First, on line 51, the constructor of `Circle` is called without any explicit arguments, which causes the special value `undefined` to be passed as default argument. Second, line 52 shows a `for-in` loop, in which the variable `a` iterates through the names of properties in the object pointed to by parameter `c`. Line 53 also provides several examples of a *dynamic property expression*. For example, the dynamic property expression `c[a]` on that line refers to the property of `c` named by the value of `a`. Together, these reflective features have the effect of copying all property values from the argument `c` into the corresponding property of the newly created object, except for `radius`, which is also multiplied by two. These features pose some challenges for refactoring:

- Applying RENAME to `this.radius` on line 4 is problematic because of the `for-in` loop and dynamic property expression in `dble`. For example, renaming the property expression to `this.rad` would require changing the string constant `"radius"` on line 53 in order to preserve behavior. In general, dynamic property expressions may

use values computed at runtime, which would thwart any static analysis. In order to ensure that dynamic property expressions do not cause changes in program behavior when applying the RENAME refactoring, our approach (as detailed in Section 4) is to conservatively disallow the renaming of any property in any object on which properties may be accessed reflectively. Hence, in this example, we disallow renaming any of the properties in `Circle` objects.

- The names of the `drawShape` methods in `Circle` and `Rectangle` must be kept consistent, because the call on line 63 may resolve to either one of these, as we explained above. Since we now disallow renaming any of the properties in `Circle`, we must also disallow renaming `drawShape` in `Rectangle`.
- The remaining properties of `Rectangle`, i.e., `x`, `y`, `width`, and `height` can still be renamed.

2.3 ENCAPSULATE PROPERTY

In Java, the ENCAPSULATE FIELD refactoring can be used to encapsulate state by making a field `private` and redirecting access to that field via newly introduced getter and setter methods [9, page 206]. Unfortunately, JavaScript does not provide language constructs to control the accessibility of properties in objects: If a function has a reference to an object, it can access any property of that object. Such a lack of encapsulation is problematic because it leads to code that is brittle, and hard to understand and maintain.

A commonly used technique, suggested for instance in Crockford's popular textbook [5], uses local variables of constructor functions to simulate private properties. Local variables in JavaScript (i.e., variables declared using the `var` keyword) can only be accessed within the scope of the declaring function. In the case of constructor functions, local variables exist as long as the object exists, and they can only be accessed by functions defined within the constructor function itself. The basic idea of the ENCAPSULATE PROPERTY refactoring is to encapsulate state by storing values in local variables instead of properties of objects, and to introduce getter/setter methods to retrieve and modify them.

Figure 3 shows how the library of Figure 1 is changed by applying the ENCAPSULATE PROPERTY refactoring to the `width` property of `Rectangle`, with changed bits of code highlighted in gray. The `width` property was changed into a local variable on line 98, and methods `getWidth` and `setWidth` were introduced on lines 101–106.[5] Furthermore, the property expression `this.width` was replaced by a call to `getWidth` on line 114. Note that there was no need to introduce a call to `getWidth` on line 110 because the `width` variable can be accessed directly. No calls to `setWidth` need to be introduced since there are no write accesses to `width`.

[5] This is not to be confused with the new getter/setter mechanism introduced in ECMAScript 5, which only applies to object literals [7, §11.1.5].

```
82  function Circle(x, y, r, c) {
83    this.x = x;
84    this.y = y;
85    this.radius = r;
86    this.color = c;
87    this.drawShape = function(gr) {
88      gr.fillCircle(new jsColor(this.color),
89                    new jsPoint(this.x,
90                    this.y),
91          this.radius);
92    };
93  }
94
95  function Rectangle(x, y, w, h, c) {
96    this.x = x;
97    this.y = y;
98    var width = w;
99    this.height = h;
100   this.color = c;
101   this.getWidth = function() {
102     return width;
103   };
104   this.setWidth = function(w) {
105     return width = w;
106   };
107   this.drawShape = function(gr) {
108     gr.fillRectangle(new jsColor(this.color),
109                      new jsPoint(this.x, this.y),
110                      width, this.height);
111   };
112 }
113 Rectangle.prototype.getArea = function() {
114   return this.getWidth() * this.height;
115 };
```

Figure 3. The library of Figure 1(a) after applying ENCAPSULATE PROPERTY to the `width` property of `Rectangle`.

The source code of the client application in Figure 1(b) is unaffected by this refactoring because it does not access the `width` property.

Name binding preservation is a key correctness condition also for the ENCAPSULATE PROPERTY refactoring, but there are other issues as well.

Encapsulating the `width` property of `Rectangle` did not cause any problems, and all other properties of `Rectangle` can be encapsulated similarly. However, this is not the case for the properties of `Circle`. To see this, consider a situation where the `radius` property of `Circle` is encapsulated in a scenario where the library is refactored together with the modified client application of Figure 2. The `for-in` loop on line 52 in the original program in Figure 2 iterates through all properties of a `Circle` object, so the behavior of this loop changes if `radius` becomes a variable instead of a property. The multiplication in the loop is no longer executed since there is no `radius` property to be copied. The `for-in` loop will also copy the `drawShape` property, but the copied function object will continue to refer to the local variables of the original `Circle` object that was being copied. As a result, the program would continue to draw circles, but with just half the radius. The ENCAPSULATE PROPERTY refactoring should clearly be disallowed in this case. A JavaScript refactoring tool must carefully take into account how properties are accessed dynamically and prevent EN-

```
116 var r1 = new Rectangle(0, 0, 100, 200, 'red');
117 var r2 = new Rectangle(0, 0, 300, 100, 'blue');
118 r1.drawShape = r2.drawShape;
119 drawAll([r1]);
```

Figure 4. Alternative client program.

CAPSULATE PROPERTY in cases where it might lead to behavioral changes. In this particular case, a tool could conservatively disallow any of the properties of `Circle` from being encapsulated.

JavaScript allows one to dynamically assign function values to properties, which causes further complications. Suppose that we want to apply ENCAPSULATE PROPERTY to the `width` property of `Rectangle` in a situation that includes the library of Figure 1(a) and the (artificial) client program of Figure 4. The original version of the program draws a red 100-by-200 rectangle. However, if `width` is encapsulated, as shown in Figure 3, a red 300-by-200 rectangle is drawn instead. To see why, note that the function stored in property `r1.drawShape` and invoked by `drawAll` comes from `r2.drawShape`, and contains the function originally created during the constructor invocation on line 117. Hence its lexical environment stores the value 300 for `width`, and this is the value read on line 110. The height, on the other hand, is read from property `height` of object `this`; the value of `this` is always the object on which the function is invoked, here `r1`, so `this.height` yields 200.

The problem can be resolved by replacing the identifier reference `width` on line 110 by a call `this.getWidth()`. In Section 3, we define the notion of *well-scopedness* to characterize functions that act as methods of a single object, making it safe to access the encapsulated property directly. Roughly speaking, a function is well-scoped if, on every call, its receiver object is the same as the value that `this` had when the function was defined. In the presence of the client of Figure 4, `drawShape` is not well-scoped because of the assignment on line 118. Therefore, our refactoring tool knows that it must replace the identifier reference `width` on line 110 by a call to `this.getWidth`.

2.4 EXTRACT MODULE

JavaScript does not provide language constructs for modularization and relies on a single global name space for all top-level functions and variables, even those that are declared in different files. This is problematic, because it can easily lead to situations where declarations of global variables and functions in one file are clobbered by those declared in another. Fortunately, it is possible to obtain most of the benefits of a module system using closures [5, page 40].

Figure 5 shows the example program of Figure 1 after applying EXTRACT MODULE to move the `Circle` and `Rectangle` functions into a new "module" called `geometry`. The basic idea is that these previously global functions become local functions inside an anonymous function, which returns an object literal with properties `Circle` and `Rectangle` through which the functions can be invoked (lines 149– 152). This anonymous function is invoked immediately (line 153), and the result is assigned to a newly introduced global variable, `geometry` (line 120). Hence, the constructor functions are now available as `geometry.Circle` and `geometry.Rectangle`. Figure 5(b) shows how the client application of Figure 1(b) is updated, by using these "qualified names". Note that this approach has the important benefit that inside the newly introduced closure function, there is no need to refer to the `geometry` variable. For example, the name `Rectangle` on line 145 need not be qualified.

A refactoring tool must take certain precautions when applying EXTRACT MODULE. For example, observe that choosing the name `shapes` for the new module is problematic because a variable with the same name is already declared on line 161. If we were to perform the refactoring anyway, the shapes "module" would be overwritten, and the constructor calls on lines 167 and 171 would cause runtime errors since the empty array `shapes` does not have properties `Circle` or `Rectangle`.

2.5 Discussion

The examples in this section show that refactoring tools for JavaScript have to address a number of challenges that do not arise in statically typed languages such as Java. Chief among these challenges is the lack of static typing and variable declarations, and the use of reflective constructs such as `for-in` loops. We address these challenges with a number of query operations defined on top of a pointer analysis framework. We present the framework and its queries in Section 3 and put them to work in Section 4 by specifying the refactorings introduced in this section in more detail.

3. A Framework for Refactoring with Pointer Analysis

In this section, we develop the technical machinery needed to precisely specify and implement refactorings like the ones described in the previous section. We first describe a set of basic queries to be provided by an underlying pointer analysis such as the one discussed in Section 5. Then, we motivate the analysis questions a refactoring tool needs to answer by taking a closer look at some of the issues illustrated above, and we show how to crystallize them into reusable queries that can be implemented on top of the basic query interface. Section 4 will demonstrate how these queries are in turn used to give detailed specifications for several refactorings.

3.1 Basic Queries

As the foundation of our framework, we assume a pointer analysis that defines a finite set $\mathbb{L}$ of *object labels* such that every object at runtime is represented by a label. We assume that $\mathbb{L}$ includes labels to represent environment records [7, §10.2.1]. For technical reasons, we require that if an object

```
var geometry = (function(){
  function Circle (x, y, r, c) {
    this.x = x;
    this.y = y;
    this.radius = r;
    this.color = c;
    this.drawShape = function (gr) {
      gr.fillCircle(new jsColor(this.color),
                    new jsPoint(this.x,this.y),
                    this.radius);
    };
  }

  function Rectangle (x, y, w, h, c) {
    this.x = x;
    this.y = y;
    this.width = w;
    this.height = h;
    this.color = c;
    this.drawShape = function (gr) {
      gr.fillRectangle(new jsColor(this.color),
                       new jsPoint(this.x,this.y),
                       this.width, this.height);
    };
  }
  Rectangle.prototype.getArea = function() {
    return this.width * this.height;
  };

  return {
    Circle : Circle,
    Rectangle : Rectangle
  };
})();
```

(a)

```
function r(n) { return Math.round(Math.random() * n); }

function drawAll(shapes) {
  var gr = new jsGraphics(document.getElementById("canvas"));
  shapes.map( function(s) { s.drawShape(gr); });
}

var shapes = [];
for (var i = 0; i < 500; i++) {
  var o = new jsColor().rgbToHex(r(255), r(255), r(255));
  switch(r(2)) {
    case 0:
      shapes[i] =
        new geometry.Circle(r(500),r(500),r(50), o);
      break;
    case 1:
      shapes[i] =
        new geometry.Rectangle(r(500),r(500),r(50), r(50), o);
      alert(shapes[i].getArea());
      break;
  }
}
drawAll(shapes);
```

(b)

Figure 5. The example program of Figure 1 after applying EXTRACT MODULE to `Circle` and `Rectangle`.

label represents an object allocated by a particular `new` expression, then all objects represented by that label are allocated by that expression. Similarly, a single object label cannot represent two function objects associated with different textual definitions.

We say that a set L of object labels over-approximates a set O of runtime objects if every object $o \in O$ is represented by some $l \in L$. For brevity, we will use the term *function definition* to mean "function declaration or function expression" and *invocation expression* to mean "function call expression or `new` expression".

The pointer analysis should provide the following queries:

objects For any expression e in the program, $\mathit{objects}(e) \subseteq \mathbb{L}$ over-approximates the set of objects to which e may evaluate, including objects arising from `ToObject` conversion [7, §9.9]. For a function declaration f, $\mathit{objects}(f)$ over-approximates the set of function objects that may result from evaluating f.

scope For any function definition or `catch` clause e, $\mathit{scope}(e) \subseteq \mathbb{L}$ over-approximates the set of environment records corresponding to e at runtime.[6] We additionally define $\mathit{scope}(e) := \mathit{objects}(e)$ for any `with` expression e.

proto For any object label ℓ, $\mathit{proto}(\ell) \subseteq \mathbb{L}$ over-approximates the possible prototype objects of the runtime objects ℓ represents. We write $\mathit{proto}^+(L)$ for the set of transitive prototypes of $L \subseteq \mathbb{L}$ as determined by this query.

props For any object label ℓ, $\mathit{props}(\ell) \subseteq \mathbb{L}$ over-approximates the set of objects that could be stored in properties of ℓ (excluding internal properties).

mayHaveProp, *mustHaveProp* For any object label ℓ and property name p, $\mathit{mayHaveProp}(\ell, p)$ should hold whenever any object represented by ℓ may have a property p; $\mathit{mustHaveProp}(\ell, p)$, conversely, should only hold if every object represented by ℓ has a property p at all times (for instance if ℓ represents an environment record and p is a local variable declared in that environment).

arg, *ret* For an object label ℓ and a natural number i, $\mathit{arg}(\ell, i)$ over-approximates the set of objects that may be passed as the ith argument (or the receiver in case $i = 0$) to any function labelled by ℓ. Similarly, $\mathit{ret}(\ell)$ over-approximates the set of objects that may be returned from ℓ.

builtin Given the name n of a built-in object as specified in the language specification [7, §15], $\mathit{builtin}(n)$ returns the corresponding object label. The object label of the global

[6] Observe that $\mathit{scope}(f)$ for a function definition f is not necessarily the same as $\mathit{objects}(f)$: the former approximates environment records, the latter approximates function objects.

object will be denoted as *global*. We also define

$$\begin{array}{lcl} \mathit{apply} & := & \mathit{builtin}(\texttt{Function.prototype.apply}) \\ \mathit{bind} & := & \mathit{builtin}(\texttt{Function.prototype.bind}) \\ \mathit{call} & := & \mathit{builtin}(\texttt{Function.prototype.call}) \end{array}$$

3.2 Visited and Base Objects

Many preconditions deal with name binding. Any refactoring that introduces, renames or removes properties risks causing *name capture*, i.e., situations where a property expression refers to a different object in the refactored program. Two key concepts are needed when formulating preconditions to avoid name capture: the *visited objects* of a property expression, and its *base objects*.

Property lookup in JavaScript is, in most circumstances, prototype based. This means that when evaluating a property expression $e.x$, the property x is first looked up on the object o_1 that e evaluates to; if o_1 does not have a property of this name, its prototype object o_2 is examined, and so on. Eventually, an object o_n is encountered that either has a property x, or does not have a prototype object (in which case the lookup returns the `undefined` value). We describe this process by saying that the lookup of $e.x$ *visits* objects $o_1, \ldots, o_n$; if the property is ultimately found on object o_n, we call o_n the *base object* of the lookup.

To see how these concepts are useful for specifying refactorings, consider the case of a refactoring that adds a property y on some object o. This refactoring needs to ensure that o is not among the objects that any existing property expression $e.y$ may visit. Otherwise, the base object of an evaluation of that expression could change, possibly altering program behavior.

The usual purpose of adding a new property y to an existing object is to rewrite property expressions that used to resolve to some property x on that object so that they now instead resolve to y. For instance, ENCAPSULATE PROPERTY rewrites `this.width` on line 26 of Figure 1 into `this.getWidth` on line 114 of Figure 3 to make it resolve to the newly introduced getter function. To prevent the refactored property expression from being resolved with the wrong base object or from overwriting an existing property, we have to require that a lookup of `this.getWidth` at this position in the original program would come up empty-handed, that is, that none of the visited objects of the property expression has a property `getWidth`. This is indeed the case in this example because no property `getWidth` is defined anywhere in Figure 1.

The same considerations apply to the lookup of local and global variables: global variables are just properties of the global object, while local variables can be viewed as properties of environment records. The concepts of visited objects and base objects can hence be extended to identifier references in a straightforward manner as shown in the accompanying technical report [8].

To underscore this commonality, we introduce the umbrella term *access* to refer to both identifier references (like `r` on line 4 of Figure 1) and property expressions, including both *fixed-property* expressions like `s.drawShape` on line 63 of Figure 2 and dynamic ones like `nc[a]` on line 53 of Figure 2.[7] Identifier references and fixed-property expressions are called *named accesses*.

An over-approximation $\mathit{possiblyNamed}(p)$ of all accesses in the program that possibly have name p in some execution, and an under-approximation $\mathit{definitelyNamed}(p)$ of accesses that definitely have name p in every execution can be computed based on purely syntactic information, although pointer analysis may provide additional information that can, e.g., be used to determine that a dynamic property access is always used as an array index and hence cannot have a non-numeric property name.

Given the basic queries introduced in Section 3.1, it is not hard to define queries *visited* and *base* to over-approximate visited and base objects of accesses.

For a property expression $e.x$, for instance, $\mathit{visited}(e.x)$ can be computed as the smallest set $L_v \subseteq \mathbb{L}$ satisfying the following two conditions:

1. $\mathit{objects}(e) \subseteq L_v$;
2. if $e.x$ is in rvalue position, then for every $\ell \in L_v$ with $\neg\mathit{mustHaveProp}(\ell, x)$ we must have $\mathit{proto}(\ell) \subseteq L_v$.

The proviso of the second condition accounts for the fact that deletion of and assignment to properties does not consider prototypes.

The definition of *visited* for identifier references is similar, using *scope* to obtain the relevant environment records.

To over-approximate the set of base objects, we first define a filtered version of *visited* as follows:

$$\mathit{visited}(a, x) := \{\ell \in \mathit{visited}(a) \mid \mathit{mayHaveProp}(\ell, x)\}$$

This discards all object labels that cannot possibly have a property x from $\mathit{visited}(a)$. For a named access a with name x in rvalue position, we then define $\mathit{base}(a) := \mathit{visited}(a, x)$, whereas for a dynamic property access or an access in lvalue position we set $\mathit{base}(a) := \mathit{visited}(a)$.

It will also be convenient to have a query $\mathit{lookup}(e, x)$ that simulates local variable lookup of an identifier x at the position of the expression e, and approximates the set of environment records or objects on which x may be resolved. This query can be implemented by traversing the function definitions, `with` blocks and `catch` clauses enclosing e, and then using *scope* and *mayHaveProp* to find possible targets.

3.3 Related Accesses

When renaming an access, it is important to determine which other accesses in the program refer to the same property. This is not a well-defined question in general: a given access

[7] The technical report [8] generalizes the concept of accesses even further, but for expository purposes we refrain from doing so here.

may at different times be looked up on different base objects and even refer to different property names, so two accesses may sometimes refer to the same property name on the same object, while at other times they do not. In general, we can only determine whether two accesses must *always* refer to the same property, or whether they may *sometimes* do so.

Must-alias information is not very useful for renaming, as explained in Section 2: when renaming `this.drawShape` on line 6 of Figure 1, we also have to rename `s.drawShape` on line 33, even though it does not necessarily refer to the same property. But if we rename `s.drawShape`, we also have to rename any access that may refer to the same property as *that* access, viz., `this.drawShape` on line 6 and `this.drawShape` on line 19.

This example suggests that we have to close the set of accesses to rename under the may-alias relation. More precisely, let us call two accesses a_1 and a_2 *directly related* if their base object may be the same and they may refer to the same property name. The set $\mathit{related}(a_1)$ of accesses *related* to a_1 is then computed as the smallest set R satisfying the following two closure conditions:

1. $a_1 \in R$;
2. for every $a \in R$, if a' is an access such that a and a' are directly related, then also $a' \in R$.

When renaming a_1 we also rename all accesses it is related to. We have argued above why it is necessary to include all related accesses in the renaming. On the other hand, it is also sufficient to just rename these accesses: if any access a' may at runtime refer to the same property as some renamed access a, then a and a' are directly related and hence a' will also be renamed. The set of related accesses thus represents a family of properties that have to be refactored together.

3.4 Initializing Functions

The ENCAPSULATE PROPERTY refactoring looks similar to the ENCAPSULATE FIELD refactoring for languages like Java and C#, but the very liberal object system of JavaScript allows for subtle corner cases that the refactoring needs to handle. While it is common in JavaScript to make a distinction between normal functions and constructor functions that are only used to initialize newly created objects, this distinction is not enforced by the language.

Any function f can either be invoked through a `new` expression `new` $f(\ldots)$, in which case the receiver object is a newly created object, or through a function invocation, in which case the receiver object is determined from the shape of the invocation: for an invocation of the form $e.f(\ldots)$, the receiver object is the value of e; for an unqualified invocation $f(\ldots)$, the receiver object is usually the global object.

We capture the notion of a function behaving "like a constructor" by saying that a function *initializes* an object o if it is invoked precisely once with that object as its receiver, and this invocation happens before any of o's properties are accessed. For instance, function `Rectangle` in Figure 1 initializes all of the objects created on line 44 by invoking `new Rectangle(...)`.

If a function is only ever invoked using `new` and never invoked reflectively or using a normal function invocation, it obviously initializes all objects created by these `new` invocations. This provides an easy way to approximate the set of objects that are initialized by a function. Let us first define an over-approximation of the set of possible callees of an invocation expression c by $\mathit{callees}(c) := \mathit{objects}(c_f)$ where c_f is the part of c containing the invoked expression. Now, given a function definition f, an under-approximation $\mathit{initializes}(f)$ of the set of objects that f initializes can be determined by ensuring the following:

1. f is only invoked through `new`, that is
   (a) No function/method call c has
   $$\mathit{callees}(c) \cap \mathit{objects}(f) \neq \emptyset.$$
   (b) f is not invoked reflectively, i.e.,
   $$\mathit{args}(\mathit{apply}, 0) \cap \mathit{objects}(f) = \emptyset,$$
   and similarly for *bind* and *call*.
2. For any `new` expression n with
   $$\mathit{callees}(n) \cap \mathit{objects}(f) \neq \emptyset$$
   we have
   $$\mathit{callees}(n) \subseteq \mathit{objects}(f)$$
   This ensures that n definitely calls f.

The first condition ensures that f is invoked at most once on each receiver object, and the second condition ensures that it is invoked at least once. If both conditions hold, f initializes all its receiver objects, so we can set $\mathit{initializes}(f) := \bigcup_{\ell \in \mathit{objects}(f)} \mathit{arg}(\ell, 0)$; otherwise, we conservatively set $\mathit{initializes}(f) := \emptyset$.

3.5 Well-scopedness

Just as there are no genuine constructors in JavaScript, there are no real methods either. Although it is common to think of a function stored in a property of an object o as a method of o that is only invoked with o as its receiver, this is not enforced by the language, and such a "method" can, in fact, be invoked on any object. As shown in Figure 4 this leads to problems when encapsulating properties.

We capture the notion of a function behaving "like a method" by the concept of *well-scopedness*. A function f is *well-scoped* in a function g if f is defined within g and whenever an execution of g on some receiver object o evaluates the definition of f, yielding a new function object f_o, then this implies that f_o is always invoked with o as its receiver. If g additionally initializes all objects on which it is

```
177 function A(g) {
178     if (g)
179         this.f = g;
180     else
181         this.f = function() {};
182 }
183
184 var a = new A(), b = new A(a.f);
185 b.f();
```

Figure 6. Example program to illustrate the approximation of well-scopedness.

invoked, then f is guaranteed to behave like a method on these objects.

To prove that a function definition f is well-scoped in g, as expressed by the query *wellscoped*(f, g), it suffices to check the following conditions:

1. f is a direct inner function of g.
2. f is only assigned to properties of the receiver of g: whenever the right-hand side e_r of a simple assignment may evaluate to f (i.e., *objects*$(e_r) \cap$ *objects*$(f) \neq \emptyset$), the sole intra-procedural reaching definition of e_r is f itself, and the left-hand side of the assignment is a property expression of the form `this`.p (for some identifier p).
3. f is only invoked on the object in whose property it is stored: any invocation expression c that may call f must be of the form $e.p(\ldots)$, and *mayHaveProp*(o, p) is false for every $o \in$ *proto*$^+$(*objects*(e)).
4. f is not invoked reflectively (cf. condition 1b in the definition of *initializes*).

The second condition is motivated by considering the example program in Figure 6. The function stored in `a.f` is not well-scoped in `A`: the receiver of `A` at the point where the function is defined is `a`, yet when it is called through `b.f` the receiver object is `b`. This non-well-scopedness results from the assignment in line 179 and is detected by condition 2.

3.6 Intrinsics and Reflective Property Access

A number of intrinsic properties are treated specially by the runtime system, the browser, or the standard library in JavaScript, for instance the `length` property of array objects or the `src` property of HTML image objects. Refactorings must not attempt to modify these properties. We hence need a query *intrinsic* so that *intrinsic*(ℓ, p) holds whenever p is an intrinsic property on an object labelled by ℓ. This query can be defined in terms of *builtin*, consulting the relevant standards [7, 29].

Several standard library functions access properties of their argument objects in a reflective way: for instance, `Object.keys` returns an array containing the names of all properties of its argument. To make it possible for refactorings to check for this kind of usage, we need a query *reflPropAcc* such that *reflPropAcc*(ℓ) holds whenever a property of an object labelled by ℓ may be accessed reflectively by one of these functions. This query can be defined in terms of *builtin*, *arg*, *ret* and *props*.

Finally, queries *builtin* and *arg* also make it possible to conservatively determine whether a program uses dynamically generated code by checking whether the built-in function `eval` and its various synonyms are ever invoked, and whether the intrinsic property `innerHTML` is assigned to. Our refactoring specifications assume that such a check is performed first and a warning is issued if a use of any of these features has been detected.

4. Specifications of Three Refactorings

We will now give detailed specifications of the refactorings RENAME, ENCAPSULATE PROPERTY and EXTRACT MODULE that were informally described in Section 2.

Each specification describes the input to the refactoring, the preconditions that have to be fulfilled in order for the refactoring to preserve program behavior, and the transformation itself. The preconditions are formulated in terms of the queries introduced in the previous section.

We also provide a brief informal justification of the preconditions.

4.1 RENAME

Input A named access a and a new name y.

Overview The refactoring renames a and its related accesses to y.

Definitions Let $B := \bigcup_{r \in \mathit{related}(a)} \mathit{base}(r)$; this set labels all objects that are affected by the renaming. Let x be the name part of the access a.

Preconditions

1. x is not an intrinsic property on B:

$$\forall \ell \in B\colon \neg \mathit{intrinsic}(\ell, x)$$

2. Every access to be renamed definitely has name x:

$$\mathit{related}(a) \subseteq \mathit{definitelyNamed}(x)$$

3. The accesses in *related*(a) can be renamed to y without name capture:

$$\forall r \in \mathit{related}(a)\colon \mathit{visited}(r, y) = \emptyset$$

   In this case, we will also say that y is *free* for *related*(a).

4. y does not cause name capture on B, that is:
   (a) Existing accesses are not captured:

$$\forall r \in \mathit{possiblyNamed}(y)\colon \mathit{visited}(r) \cap B = \emptyset$$

   (b) y is not an intrinsic property on B:

$$\forall \ell \in B\colon \neg \mathit{intrinsic}(\ell, y)$$

(c) Properties of the objects in B must not be accessed reflectively, that is:

i. For any `for-in` loop with loop expression e it must be the case that $B \cap \mathit{objects}(e) = \emptyset$.

ii. We must have $\forall \ell \in B\colon \neg \mathit{reflPropAcc}(\ell)$.

Transformation Rename every access in $\mathit{related}(a)$ to y.

Justification Precondition 2 prevents the renaming if it could affect a computed property access whose name cannot be statically determined.

Preconditions 3 and 4a ensure that accesses in the refactored program resolve to the same property at runtime as in the original program: by 3, an access renamed from x to y is not captured by an existing property y; by 4a, an existing access named y is not captured by a property renamed from x to y.

Preconditions 1 and 4b ensure that the renaming does not affect properties that have special meaning in the language; for instance, renaming the `prototype` of a function or the `length` property of an array should not be allowed.

Finally, precondition 4c ensures that none of the objects whose properties may be affected by the refactoring have their properties examined reflectively.

4.2 ENCAPSULATE PROPERTY

Input A fixed-property expression a.

Overview This refactoring identifies a function c that initializes all base objects of a and its related accesses, and turns the property accessed by a into a local variable of c.

Any accesses to the property from within the function c can be turned into accesses to the local variable if they happen from inside well-scoped functions; otherwise they might refer to the wrong variable as seen in Section 2. Accesses from outside c are handled by defining getter and setter functions in c and rewriting accesses into calls to these functions.

The preconditions identify a suitable c, determine how to rewrite accesses, and check for name binding issues.

Definitions Let x be the name part of a, and let g and s be appropriate getter and setter names derived from x.

Let $B := \bigcup_{r \in \mathit{related}(a)} \mathit{base}(r)$; this is the set of objects whose properties named x we want to encapsulate.

Preconditions

1. There is a function definition c with $B \subseteq \mathit{initializes}(c)$.

   The getter and setter functions are introduced in c; since c is invoked on every affected object before any of its properties are accessed, we can be sure that these functions are in place before their first use.

2. The affected objects do not appear on each other's prototype chains, i.e.,

$$\neg\exists b_1, b_2 \in B : b_2 \in \mathit{proto}^+(b_1)$$

3. Every access in $\mathit{related}(a)$ is either a fixed-property expression or an identifier reference. (The latter can only happen if a `with` statement is involved.)

4. There is a partitioning $\mathit{related}(a) = A_i \uplus A_g \uplus A_s$ such that:

   (a) Every $a \in A_i$ is of the form `this.x`, it is not an operand of `delete`, and its enclosing function definition f is well-scoped in c, i.e. $\mathit{wellscoped}(f, c)$.

   These are the accesses that will be replaced by identifier references `x`.

   (b) No $a \in A_g$ is in an lvalue position.

   These accesses can be turned into invocations of the getter function.

   (c) Every $a \in A_s$ forms the left-hand side of a simple assignment.

   These accesses can be turned into invocations of the setter function.

5. Properties of B must not be accessed reflectively (cf. precondition 4c of RENAME).

6. Naming checks:

   (a) A_i can be refactored without name capture:

$$\forall a \in A_i\colon \mathit{lookup}(a, x) \subseteq \{\mathit{global}\}$$

   (b) The declaration of the new local variable `x` in c does not capture existing identifier references.

$$\forall a \in \mathit{possiblyNamed}(x)\colon \mathit{visited}(a) \cap \mathit{scope}(c) = \emptyset$$

   (c) x is not an intrinsic property on B:

$$\forall \ell \in B\colon \neg \mathit{intrinsic}(\ell, x)$$

7. If $A_g \neq \emptyset$ then g must be free for A_g and must not cause name capture on $\mathit{initializes}(c)$ (cf. preconditions 3 and 4 of RENAME). Similarly, if $A_s \neq \emptyset$ then s must be free for A_s and must not cause name capture on $\mathit{initializes}(c)$.

Transformation Insert a declaration `var x` into c. Insert a definition of the getter function into c if $A_g \neq \emptyset$, and similarly for A_s and the setter function. Replace accesses in A_i with `x`, accesses in A_g with invocations of the getter, in A_s with invocations of the setter.

Justification This refactoring converts properties of objects into bindings in environment records. The preconditions ensure that property accesses can be rewritten into accesses to the corresponding local variable binding, while preventing any changes to other accesses to properties or local variables that do not participate in the refactoring.

Consider a runtime object o labeled by some $\ell \in B$. By condition 1, there is precisely one invocation of c on o, which creates an environment record ρ_o. In the refactored

program, this environment record contains a binding for a local variable x, which is captured by the getter and setter functions stored in properties g and s of o.

Consider now a property access a_x in the original program that accesses property x of object o. This means that $a_x \in \mathit{related}(a)$, so condition 4 ensures that a_x is in one of A_i, A_g and A_s. In the two latter cases, the property access will be rewritten into an invocation of the getter method g or the setter method s on o.

By condition 7 this invocation will not be captured by another method of the same name, and by condition 2 it will not be captured by the accessor methods of another refactored object. By condition 1, g and s are already defined, and by condition 7 they are guaranteed not to have been overwritten in the meantime, hence the accessor functions set up by c are executed, accessing the correct binding in ρ_o.

If $a_x \in A_i$, the property access is refactored to a simple identifier reference x. We know from condition 4a that a_x must occur in some function definition f, which is well-scoped in c, and that it must be of the form `this.x`. Hence f is, in fact, invoked with o as receiver, which by the definition of well-scopedness means that the invocation of c whose bindings are captured by f also has receiver o. In other words, f captures the bindings of ρ_o. Condition 6a ensures that the identifier reference x in the refactored program is not captured by any other local variable, and hence accesses the binding of x in ρ_o as desired.

The requirement about a_x not being an operand of `delete` is purely technical: local variable bindings cannot be deleted in JavaScript.

Since the set of properties of o has changed in the refactored program, any code that reflectively accesses properties of o or the set of property names of o may change its behavior; conditions 3, 5 and 6c guard against this. Finally, condition 6b ensures that no existing local variable bindings are upset by the newly introduced local variable x in c.

Remarks Note that condition 4 makes it impossible to refactor accesses like `++e.x` that both read and write the encapsulated property, unless they can be replaced by an identifier reference. It is straightforward to extend the refactoring to take care of such accesses, at the cost of a slightly more complicated transformation involving both getter and setter invocations in the same expression [8].

4.3 Extract Module

Input Contiguous top-level statements $s_1, \ldots, s_m$ containing a set $P = \{p_1, \ldots, p_n\}$ of identifiers to extract and an identifier M to be used as module name.

Overview The global variables $p_1, \ldots, p_n$ are turned into properties of a newly declared global module variable M. Schematically, the transformation performed by the refactoring is as follows:

```
s1;                          var M = (function() {
⋮                    ⇒         var p1, ..., pn;
sm;                            s1; ... sm;
                               return {
                                 p1: p1, ..., pn: pn
                               };
                             })();
```

We refer to the code defining M as the *module initialization code*. To reason about the correctness of the transformation, it is helpful to partition program execution into three phases: before, during and after execution of the initialization code. Being a top-level statement, the module initialization code is executed only once.

None of the variables in P must be accessed before module initialization since the module M containing them has not been defined yet. After module initialization, on the other hand, they can be accessed as properties of M, i.e., $M.p_1, \ldots, M.p_n$. It is clearly not possible to access them in this way during module initialization (M is, after all, not defined yet), but we can instead access the corresponding local variables $p_1, \ldots, p_n$ if they are in scope.

Closures created during module initialization may still be able to access a local variable even after module initialization has finished. This should, however, be avoided unless it can be proved that the variable is never assigned to after module initialization: if not, the local variable p_i and the property $M.p_i$ may have different values, which could change program behavior.

The preconditions determine a set Q of accesses that have to be converted into property accesses of the form $M.p_i$, and a set U of accesses that can use the local variables of the module. The preconditions also prevent access to module variables before initialization and name binding issues.

Definitions Let S be the set of all accesses appearing in the statements $s_1, \ldots, s_m$, and let $I \subseteq S$ be the accesses that are not nested inside functions. Accesses in I are thus guaranteed to only be evaluated during module initialization.

Let I^* be an over-approximation of the set of all accesses that may be evaluated before or during module initialization. This can be obtained by building a transitive call-graph of all top-level statements up to s_m, using query *callees* to determine possible callees of invocations. Finally, let C contain all accesses in the program except those in I^*. Accesses in C are thus guaranteed only to be evaluated after module initialization is complete.

For $p \in P$, we define A_p to be the set of accesses that may refer to the global variable p, and $A_P := \bigcup_{p \in P} A_p$. We define $\mathit{mutable}(p)$ to hold if A_p contains a write access that does not belong to I, i.e., if p may be written after module initialization is complete.

Preconditions

1. Any access that *may* refer to some property in P *must* refer to that property, i.e., for every $p \in P$ and $a \in A_p$:

$$a \in \mathit{definitelyNamed}(p) \wedge \mathit{visited}(a, p) = \{\mathit{global}\}$$

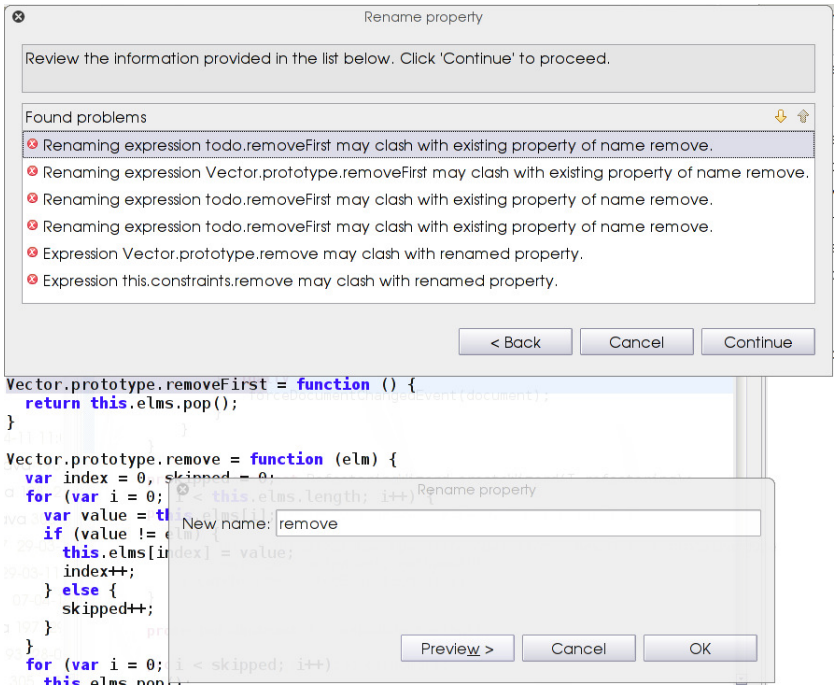

Figure 7. The refactoring plug-in for Eclipse. The user has attempted to rename `Vector.prototype.removeFirst` to `remove`, which the tool correctly determines would clash with an existing property of the same name.

2. There is a partitioning $A_P = Q \uplus U$ as follows:
   (a) $Q \subseteq C$
   (b) M is free for every $q \in Q$ (cf. precondition 3 of RENAME).
   (c) For every $u \in U$ referring to $p \in P$, the following holds:
      i. $u \in I \vee (u \in S \wedge \neg \mathit{mutable}(p))$
      ii. u is an identifier reference.
      iii. $\mathit{lookup}(u, p) \subseteq \{\mathit{global}\}$.
3. M does not cause name capture on *global* (cf. precondition 4 of RENAME).
4. No $p \in P$ is an intrinsic on *global*:

$$\forall \ell \in B\colon \neg \mathit{intrinsic}(\ell, p)$$

Transformation Replace $s_1, \ldots, s_m$ with the definition of module M as shown above; qualify accesses in Q with M.

Justification The refactoring introduces a new global variable M and removes the global variables $p_1, \ldots, p_n$. Condition 3 ensures that no existing access to a variable M is captured by the newly introduced module variable, and that the set of global variables is not examined reflectively. Condition 4 ensures that none of the global variables to be modularized has special semantics. It should, for instance, be impossible to extract the global variable `window` into a module.

The remaining preconditions ensure that accesses to global variables $p_1, \ldots, p_m$ can be consistently refactored. Condition 1 requires that any access either must definitely refer to some $p \in P$, or must not refer to any variable in P. Condition 2a checks that accesses in Q, which are to be qualified with a reference to M, are only evaluated after the module is defined. For the same set of accesses, condition 2b ensures that the reference to M that will be inserted by the refactoring cannot be captured by an existing variable M.

Finally, condition 2c makes sure that every access $u \in U$, which used to refer to one of the global variables $p \in P$, can directly access the local variable this variable has been turned into. Sub-condition 2(c)i requires that u is either only evaluated during module initialization, or that it refers to an immutable module member and is lexically nested within the module definition. Either way it can access module members without qualification. Sub-condition 2(c)ii rules out the somewhat subtle case of an access of the form $e.p$, where e evaluates to the global object, but may have side effects; such an access cannot simply be turned into an identifier reference p, as this would suppress the side effects of e. Sub-condition 2(c)iii ensures that no existing local variable will capture the refactored access u.

5. Implementation

We have implemented a refactoring tool in Java that offers the refactorings described in Section 4. The tool is integrated

as a plug-in into Eclipse as shown in Figure 7.[8] In this section, we will describe the pointer analysis that underlies the implementation of the framework that we presented in Section 3.

We first derive a flow graph from the source code of the original program, similar to the one used in the TAJS program analysis [18]. From this flow graph, we create a def-use graph that abstracts away control flow and `with` statements. We then run a pointer analysis using standard techniques, with lattice and constraints that are reminiscent of the ones used in Gatekeeper [12] (although without using Datalog). The use of a def-use graph captures a small amount of flow sensitivity, similar to what SSA-form has been shown to contribute to a flow-insensitive analysis [16].

For context sensitivity, we experimented with both k-CFA and object sensitivity (i.e., using the value of `this` as the context), and found object sensitivity to be the most effective. The analysis uses heap specialization (i.e., some object labels include a context component) and a simple widening function to ensure termination when combined with object sensitivity.

To obtain a useful modeling of arrays, we introduce a special property name *NumberProperty* representing all properties whose name is a number (i.e., array entries). For dynamic property expressions where the property name is definitely a number, the analysis reads/writes the *NumberProperty* of the receiver; otherwise, it conservatively reads/writes all of its properties.

Several built-in functions (such as `call` and `apply`) are supported by means of specialized transfer functions. All other built-in functions are modelled by simple JavaScript mock-up functions that we include in the analysis.

We model the HTML DOM and some other browser features using a special object label *DOM*. Some global variables, such as `document`, are initialized to refer to *DOM*. Moreover, we conservatively assume that (1) any property of *DOM* may point to *DOM*, (2) any function reachable from *DOM* may be invoked with *DOM* as the `this` argument and any number of actual arguments that all may point to *DOM*, and (3) if *DOM* is invoked as a function, it stores all its arguments as properties on *DOM*, and returns *DOM*. Rules 2 and 3 together take care of event handlers being registered on HTML elements. We avoid many of the challenges that arise with the more detailed modeling used in TAJS [20] by using a relatively simple abstract domain.

Given this basis, the queries of the framework of Section 3 are straightforward to implement, as are the refactorings themselves.

[8] Note that this is purely a UI-level integration; the underlying analysis and the code for precondition checking and program transformation is independent of Eclipse.

6. Evaluation

To gain some insight into the practical applicability and usefulness of our approach, we have evaluated our refactoring tool on a collection of existing JavaScript programs.

In situations where the tool determines that a requested refactoring can be performed, the refactoring preconditions ensure that it is safe to perform the refactoring, without changing the behavior of the program. When a refactoring attempt is rejected by the tool, either the refactoring would in fact change the behavior of the program, in which case the answer given by the tool is correct, or the rejection is caused by the analysis being too conservative. In the latter case, the imprecision may be in the refactoring preconditions that are defined in terms of our queries (Section 4), in the definition of the derived queries on top of the basic ones (Section 3.2–3.6), or in the underlying pointer analysis that we employ to implement the basic queries (Section 5). To quantify how often these situations occur, we aim to answer these research questions:

Q1: How often is a refactoring rejected because its preconditions are too conservative?

Q2: How often is a refactoring rejected because a derived query is defined too conservatively?

Q3: How often is a refactoring rejected because of imprecision in the underlying pointer analysis?

For the RENAME refactoring, it is also relevant how it performs compared to the naive alternative of simply using search-and-replace through the program source code:

Q4: How often does our RENAME refactoring give a different outcome than syntactic search-and-replace as performed in syntax-directed editors?

We collected a suite of benchmark programs and designed a set of experiments for each of the refactorings specified in Section 4 to evaluate them with regard to these questions.

Table 1 shows an overview of our evaluation results, explained in more detail below: for every refactoring, the table shows the total number of attempted refactorings on our benchmarks in column "total applications", with the number of successful applications in the next column; we partition the set of rejected applications according to our research questions into cases where overly strict preconditions prevented the application of an otherwise unproblematic refactoring, cases where imprecise derived queries were an obstacle, cases where the underlying pointer analysis itself was at fault, and finally cases where the rejection was indicative of a real danger of unsoundness.

We will now first give an overview of our benchmark collection, then present detailed evaluation results for each of the refactorings, and finally summarize our findings by answering the research questions.

refactoring	total applications	successful applications	rejected applications				
			total	imprecise preconditions	imprecise queries	imprecise analysis	justified
RENAME	16612	10693	5919	0	0	669	5250
ENCAPSULATE PROPERTY	510	363	147	35	0	30	82
EXTRACT MODULE (1)	50	43	7	0	0	0	7
EXTRACT MODULE (2)	15	11	4	0	0	0	4

Table 1. Quantitative evaluation of our refactoring tool.

6.1 Benchmark Programs

We have gathered 50 JavaScript programs. Four are taken from the V8 benchmarks,[9] 23 from Chrome Experiments,[10] 18 from the 10K Apart Challenge,[11] and 5 from IE Test Drive.[12] When collecting these benchmarks, we explicitly excluded programs that our pointer analysis cannot analyze in a few minutes and ones that use non-trivial dynamic code execution (e.g., using `eval`). Four of the benchmarks use trivial dynamic code, such as `setTimeout("loop();", 50)`, which we have manually replaced by the more manageable variant `setTimeout(loop, 50)`. For 27 of the benchmarks, the tool produces a warning that they may contain assignments to the `innerHTML` property of a DOM object, which can potentially be used to run dynamically generated code, however manual inspection revealed that this is not the case in any of the programs.

Each benchmark comprises between 300 and 1700 lines of JavaScript code, and all perform non-trivial tasks. On a 3.0 GHz PC, each benchmark is analyzed in less than 4 seconds using 256 MB memory. The time required for refactoring-specific computations is negligible compared to the time taken by the pointer analysis.

6.2 RENAME

Our RENAME refactoring can rename both local variables and properties. Local variables are trivial to rename since there are no `with` statements in our benchmarks, so we focus on renaming of properties.

We have systematically applied our refactoring to every property expression and property initializer in each benchmark, with the aggregate results shown in Table 1 in the row labeled RENAME. Out of a total of 16612 attempted rename operations, 10693 were successfully applied, and 5919 were rejected by our tool. Further analysis revealed that of these rejections, 5250 were justified. Two benchmarks are responsible for the remaining 669 rejections. In *raytracer* from Chrome Experiments, there are 1062 renamable accesses but 665 of these are wrongly rejected, due to the pointer analysis being imprecise. In *flyingimages* from the IE Test Drive benchmarks, the program adds some custom properties to a DOM element, which due to our imprecise DOM model are then assumed to be intrinsic; thus our tool refuses to rename these properties. The remaining 48 benchmarks do not give rise to any RENAME-specific spurious warnings.

To evaluate how our tool compares to a simple search-and-replace performed at the level of the abstract syntax tree (AST) in a syntax directed editor, we use the equivalence classes defined by the *related* query to divide all the accesses in a benchmark into *components*. Accesses in a single component always get renamed together. Our tool distinguishes itself from simple search-and-replace tools when different components contain accesses with the same name. In particular, our tool will rename a smaller set of accesses than search-and-replace would, and if one component can be renamed while another cannot (e.g., an access in it may refer to an intrinsic property), search-and-replace would change the program's behavior, whereas our approach would reject the refactoring.

The tool finds that 28 of the 50 benchmarks contain multiple renamable components with the same name, and 19 contain same-name components where some can be renamed and others are correctly rejected (18 benchmarks fall into both categories). Overall, our tool succeeds in renaming 1567 components, with 393 of them having a name in common with another component in the same program.This indicates that our RENAME refactoring will often be more precise than search-and-replace in practice.

To summarize, RENAME leads to smaller source code transformations than search-and-replace in about 25% of the cases. Of the refactoring attempts that were not justifiably rejected, it issues spurious warnings in only 6% of the cases. The spurious warnings are all caused by imprecision in the pointer analysis.

6.3 ENCAPSULATE PROPERTY

We have exhaustively applied the ENCAPSULATE PROPERTY refactoring to every property expression of the form `this.x` appearing in an lvalue position inside a function that is invoked at least once in a `new` expression, with the results shown in Table 1 in the row labeled ENCAPSULATE PROPERTY.

In the 50 benchmarks, there are 510 such expressions. Our tool is able to successfully encapsulate 363 of them,

[9] `http://v8.googlecode.com/svn/data/benchmarks/`

[10] `http://www.chromeexperiments.com/`

[11] `http://10k.aneventapart.com/`

[12] `http://ie.microsoft.com/testdrive/`

ignoring warnings about assignments to `innerHTML`. In the remaining 147 cases, the tool reports a precondition failure and rejects the refactoring.

For 82 of these cases, the rejection is justified: in three cases, getter/setter methods already exist; in eight cases the encapsulated property would shadow references to a global variable; in the remaining 71 cases there is a name clash with a parameter or local variable of the enclosing function. We manually verified that these cases can be refactored successfully if the naming conflict is first resolved by renaming.

Of the 65 remaining cases, where the refactoring is rejected although it should have been successful, 35 are due to a limitation of our specification of ENCAPSULATE PROPERTY: it requires all objects on which the property is encapsulated to be initialized by the same function. In some cases, however, there are identically named properties on objects constructed by different constructors, which need to be encapsulated at the same time because there are accesses that may refer to either property. Supporting this situation seems like a worthwhile extension.

Finally, there are 30 cases where the pointer analysis yields imprecise results that cause spurious precondition violations. Of these, 19 cases could be fixed by improving the modelling of standard library array functions.

The concept of well-scopedness and the conservative analysis to determine well-scopedness described above prove to be adequate on the considered benchmarks: there are 28 cases where properties to be encapsulated are accessed from within an inner function of the constructor, and in all cases the analysis can establish well-scopedness, allowing the access to be replaced by an identifier reference instead of a getter invocation.

In summary, our tool is able to handle about 85% of the encapsulation attempts satisfactorily (not counting the justifiably rejected attempts). The remaining 15% are caused by, in about equal parts, restrictions of the specification and imprecision of the pointer analysis.

6.4 EXTRACT MODULE

The EXTRACT MODULE refactoring is difficult to evaluate in an automated fashion, since appropriate module boundaries have to be provided for every benchmark. We have performed two sets of experiments. In the first experiment, we extracted, for every benchmark, the code in each HTML `script` element into its own module; in the case of standalone benchmarks we chose source files as the unit of modularization instead. The results of this experiment are shown in Table 1 in the row labeled EXTRACT MODULE (1). In the second experiment, we manually determined a suitable modularization for a subset of our benchmarks and used our tool to perform it; again, the results are shown in Table 1 in row EXTRACT MODULE (2).

For the first experiment, the automated modularization was successfully performed on 43 out of 50 benchmarks. On the remaining seven benchmarks, the refactoring was rejected since they contain accesses to module members for which the refactoring cannot prove that they either definitely happen only during module initialization, or definitely happen only after initialization. These rejections turn out to be justified: the accesses in question are performed by event handlers registered before or during module initialization. While it is highly likely that these handlers will not fire until after initialization is complete, this is not guaranteed.

In three cases, the rejections are arguably due to the very coarse module structure imposed by this experiment. If the code that installs the event handlers is excluded from the module, the handlers are guaranteed to only fire after initialization and the refactoring can go ahead. In the remaining four benchmarks, on the other hand, event handlers are installed through HTML attributes before the handler functions are even defined, which could potentially cause races even in the original program.

For the second experiment, we randomly selected 15 benchmarks that are not already modularized and whose global variables have sufficiently descriptive names to make it easy to manually determine a possible modularization. In three of these programs, we took comments into account that already suggested a functional grouping of global functions. Our tool can perform the proposed modularization on 11 of the 15 benchmarks. The remaining four are again rejected due to potential races on event handlers.

In both experiments, our tool was thus able to handle all test cases correctly. The categorization of accesses according to whether they are evaluated before or after module initialization proved to be a valuable aid in detecting potentially subtle bugs that could be introduced by the refactoring.

6.5 Summary

Overall, the results of our evaluation are promising. Most attempted refactorings are performed successfully, and when our tool rejects a refactoring it mostly does so for a good reason. We briefly summarize our findings and answer the general research questions posed at the beginning of this section.

Q1: Rejections due to rigid preconditions Spurious rejections resulting from overly conservative preconditions are not very common: this happens in 35 out of 510−82 applications (8.2%) of ENCAPSULATE PROPERTY, and not at all for RENAME and EXTRACT MODULE.

Q2: Rejections due to derived queries The derived queries are always sufficiently precise in our experiments. For instance, ENCAPSULATE PROPERTY needs to prove well-scopedness for 28 functions, and all of them are indeed shown to be well-scoped by the algorithm described in Section 3.5.

Q3: Rejections due to imprecise pointer analysis Spurious rejections resulting from imprecision of the pointer analysis occur occasionally: 669 of 16612−5250 applications

(5.9%) of RENAME and 30 of 510−82 applications (7.0%) of ENCAPSULATE PROPERTY are rejected for this reason; and none for EXTRACT MODULE.

Q4: Improvement over naive search-and-replace For 393 out of 1567 groups of accesses that must be renamed together (25%), RENAME avoids some of the unnecessary modifications performed by AST-level search-and-replace.

These results indicate that the precision of the refactoring preconditions, the derived queries, and the pointer analysis is sufficient for practical use, and that our technique has advantages in practice compared to naive approaches.

6.6 Discussion

The validity of our evaluation may be threatened by (1) benchmark selection, (2) analysis limitations, and (3) selection of refactoring targets.

While we only consider a relatively small number of benchmarks of modest size, the programs included do demonstrate a variety of application areas, from the more numerically oriented V8 benchmarks to browser-based games and visualization programs in the other benchmark sets. They also exhibit very different programming styles, with some benchmarks making heavy use of the object system and others written in an entirely procedural style.

One notable feature of all our benchmarks is that none of them make use of a framework library such as jQuery, Prototype, or MooTools. The pointer analysis currently used in our implementation cannot tackle such libraries due to scalability issues. It is possible that the meta-programming techniques employed by some of these frameworks could lead to very imprecise analysis results that may lead to a large number of spurious rejections. In this case, it could be worthwhile to extend the analysis with special knowledge about particularly tricky framework functions.

Our analysis has certain limitations that may affect the validity of our results. In particular, our implementation only analyzes code that is reachable either from top-level statements or from the DOM. Other code does not influence the refactoring and is itself not affected by refactoring. This means that our tool cannot safely be applied to library code alone, since most of the functions in a library will be considered dead code when there is no client to invoke them. For statically typed languages, this problem can be side-stepped by assuming, for instance, every method to be an entry point, with the parameter types providing a conservative approximation of the possible points-to sets of arguments. This is not easy to do in JavaScript, and making worst-case assumptions about argument values would lead to unacceptable precision loss. All of our benchmarks are standalone applications, yet about half of them contained some amount of unused code. This indicates that the issue may indeed deserve further attention.

As a second restriction, our analysis currently does not attempt to analyze dynamically generated code. We handle this in our refactoring tool by issuing a warning if a potential use of such code is detected to alert the user of possible changes to the behavior of the program.

Finally, our pointer analysis does not currently model ECMAScript 5 getter and setter properties on object literals, but these are not used in the benchmarks anyway.

These shortcomings of the analysis, however, do not seriously jeopardize the validity of our approach, since we have been careful to introduce a clean separation between analysis and refactoring by means of the framework described in Section 3. This makes it easy to plug in a more powerful pointer analysis without having to change the specifications or implementations of the refactorings themselves.

As a final threat to validity, one might question the selection of targets on which to apply our refactoring tool. We have based our evaluation on exhaustively applying the refactorings to as many targets in the code as possible to avoid selection bias. Many of these applications would most likely not make sense in an actual development context; it is hence not clear what percentage of spurious rejections a user of our tool would experience in practice. However, the overall percentage of spurious rejections in our experimental evaluation is so low as to make it seem likely that our tool would behave reasonably in practice.

7. Related Work

Two broad categories of related work can be distinguished: previous work on refactoring in general, and work on static analysis of JavaScript programs.

7.1 Refactoring

The field of refactoring started in the early 1990s with the Ph.D. theses of Opdyke [23] and Griswold [11]. Since then, the refactoring community has focused on developing automated refactoring tools for both dynamically typed languages (e.g., [22, 24]), and for statically typed languages (e.g., [10, 27, 28]). The discussion below will focus on previous work on refactoring for dynamically typed languages.

Work by Roberts et al. on the Refactoring Browser [24] targets Smalltalk, a dynamically typed language in which some of the same challenges addressed in this paper arise. For method renaming, e.g., it becomes difficult or impossible to determine statically which call sites need to be updated in the presence of polymorphism and dynamically created messages. Unlike our approach, which is based on static pointer analysis, Roberts et al. adopt a dynamic approach to this problem, in which renaming a method involves putting a method wrapper on the original method. As the program runs, the wrapper detects sites that call the original method and rewrites those call sites to refer to the renamed method instead. The main drawback of this approach is that it relies on a test suite that exercises all call sites to be rewritten.

The Guru tool by Moore [22] provides automatic refactoring for the Self programming language. Guru takes a col-

lection of objects, which need not be related by inheritance, and restructures them into a new inheritance hierarchy in which there are no duplicated methods, in a way that preserves program behavior. Moore's algorithm is based on a static analysis of the relationship between objects and methods in the system. Unlike our work, Moore's approach does not involve the programmer in deciding what refactorings to apply and where to apply them.

Refactoring support in IDEs for JavaScript appears to be in its infancy. Eclipse JSDT [6] and the JetBrains JavaScript Editor [21] aim to provide refactoring support for JavaScript, but the current implementations are fairly naive. RENAME in the JavaScript Editor, for instance, seems to essentially just perform search-and-replace on the AST. Renaming property `x` in `Circle` in the example of Figure 1, for instance, would also rename all properties with name `x` in the jsDraw2D library that the program uses.

Two projects at the IFS Institute for Software focused on developing JavaScript refactoring plug-ins for Eclipse JSDT [2, 3], but their results do not seem to have been published and are not currently available.

7.2 Analysis for JavaScript

Several authors have pursued forms of static program analysis for JavaScript. The TAJS analysis tool by Jensen et al. [18–20] aims at detecting common programming errors in JavaScript programs. Anderson et al. [1] define a type system for a core calculus based on JavaScript along with an associated constraint-based type inference algorithm. Jang and Choe [17] use a constraint-based analysis for optimizing programs written in a restricted variant of JavaScript. The Gatekeeper tool by Guarnieri and Livshits [12] and the Actarus tool by Guarnieri et al. [13] use static analysis to enforce security policies in JavaScript programs, e.g., that a program may not redirect the browser to a new location or that untrusted information cannot flow to sensitive operations. Guha et al. [15] describe a core calculus for JavaScript and use that formalism to design a type system that statically ensures a form of sandboxing. Other work by Guha et al. [14] involves a k-CFA analysis for extracting models of client behavior in AJAX applications. The Kudzu tool by Saxena et al. [25] performs symbolic execution on JavaScript code and uses the results to identify vulnerability to code injection attacks.

Like our work, many of these analyses rely heavily on the results of a pointer analysis. For example, the TAJS tool performs a pointer analysis as part of its analysis, the optimization technique by Jang and Choe relies directly on pointer analysis, and Gatekeeper's security policies are expressed in terms of a Datalog-based pointer analysis. In all of these instances, the pointer analysis provides may-point-to information, similar to the underlying analysis in our refactoring framework. However, as we have illustrated in Sections 2 and 3, may-point-to information does not directly provide a useful abstraction for sound refactorings in JavaScript, which has motivated the higher-level concepts that appear as queries in our framework, such as the notions of relatedness and well-scopedness.

8. Conclusion

We have presented a principled approach for tool-supported refactoring for JavaScript programs. The key insight of our work is that—despite the challenging dynamic features of the JavaScript language—it is possible to capture fundamental correctness properties of JavaScript refactorings using a small collection of queries in a framework based on pointer analysis. With this framework, we have demonstrated that the complex preconditions of refactorings, such as RENAME, ENCAPSULATE PROPERTY and EXTRACT MODULE, can be expressed in a concise manner. Our experiments show that the refactoring preconditions we formulate have high accuracy. Most importantly, if a programmer's request to perform a refactoring is rejected by the tool, it is usually because the refactoring would in fact change the behavior of the program.

In future work, we will focus on advancing the scalability of the underlying pointer analysis, and we plan to provide specifications and implementations of other refactorings by way of our framework. Another direction of work is to adapt our techniques to other dynamically typed languages.

References

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *Proc. 19th European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*. Springer-Verlag, July 2005.

[2] C. Bachmann and E. Pfister. JavaScript Refactoring Eclipse Plug-in. Bachelor thesis, University of Applied Science Rapperswil, 2008.

[3] M. Balmer and R. Kühni. Refactoring Support für Eclipse JavaScript Development Tools. Diploma thesis, University of Applied Science Rapperswil, 2008.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[5] D. Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[6] Eclipse. JavaScript Development Tools. `http://wiki.eclipse.org/JSDT`.

[7] ECMA. ECMAScript Language Specification, 5th edition, 2009. ECMA-262.

[8] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported Refactoring for JavaScript. `http://www.brics.dk/jsrefactor/`, 2011. Technical Report.

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] A. Garrido and R. E. Johnson. Refactoring C with Conditional Compilation. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, October 2003.

[11] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis, University of Washington, 1991.

[12] S. Guarnieri and V. B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. 18th USENIX Security Symposium*, August 2009.

[13] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. In *Proc. 20th International Symposium on Software Testing and Analysis*, July 2011.

[14] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proc. 18th International Conference on World Wide Web*, May 2009.

[15] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *Proc. 24th European Conference on Object-Oriented Programming*, volume 6183 of *LNCS*. Springer-Verlag, June 2010.

[16] R. Hasti and S. Horwitz. Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[17] D. Jang and K.-M. Choe. Points-to Analysis for JavaScript. In *Proc. 24th Annual ACM Symposium on Applied Computing, Programming Language Track*, March 2009.

[18] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[19] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural Analysis with Lazy Propagation. In *Proc. 17th International Static Analysis Symposium*, volume 6337 of *LNCS*. Springer-Verlag, September 2010.

[20] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, September 2011.

[21] JetBrains. JavaScript Editor. `http://www.jetbrains.com/editors/javascript_editor.jsp`.

[22] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1996.

[23] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[24] D. B. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4): 253–263, 1997.

[25] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, D. Song, and F. Mao. A Symbolic Execution Framework for JavaScript. In *Proc. 31st IEEE Symposium on Security and Privacy*, May 2010.

[26] M. Schäfer, T. Ekman, and O. de Moor. Sound and Extensible Renaming for Java. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2008.

[27] F. Tip. Refactoring Using Type Constraints. In *Proc. 14th International Static Analysis Symposium*, volume 4634 of *LNCS*. Springer-Verlag, August 2007.

[28] D. von Dincklage and A. Diwan. Converting Java Classes to Use Generics. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2004.

[29] World Wide Web Consortium. Document Object Model Level 3. `http://www.w3.org/DOM/DOMTR#dom3`.