

Tool-supported Refactoring for JavaScript

Asger Feldthaus
Aarhus University

Todd Millstein
University of California, Los Angeles

Anders Møller
Aarhus University

Max Schäfer
University of Oxford

Frank Tip
IBM Research

Prepared by:
Łukasz Białek (lb277555)

Introduction

Refactoring – what is this?

Refactoring is a popular technique for **improving the structure** of existing programs while **maintaining their behavior.**

JavaScript? Why?

Refactoring for statically typed languages like Java is well developed.

Refactoring tools for JavaScript are less mature and not always can ensure that program behavior is preserved.

How it should be done

Refactoring is the process of improving the structure of software by applying behavior-preserving program **transformations**.

These transformations are typically identified by a name.

How it should be done

They also have their preconditions under which they are applicable and set of algorithm steps how to do them.

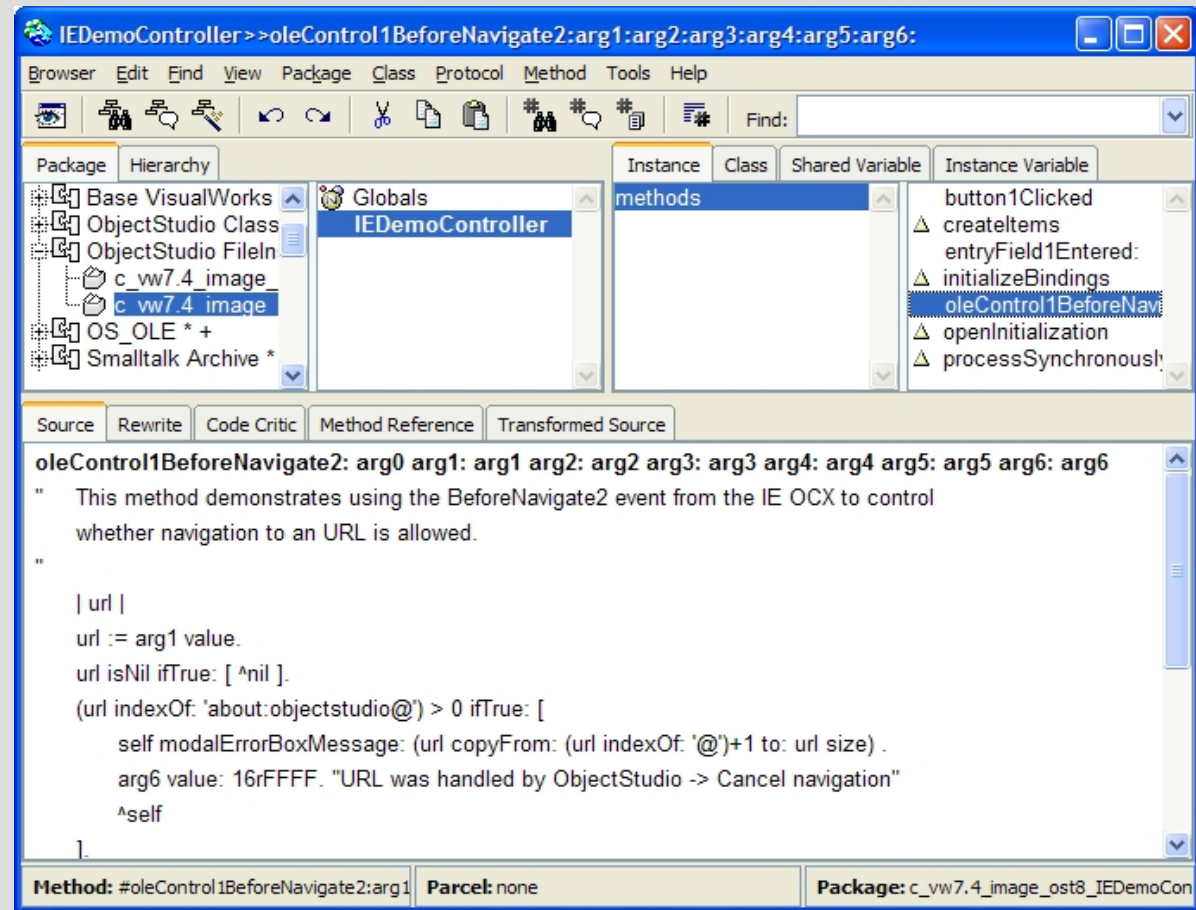
If you apply them manually, you are probably doing it wrong!

Approaches to refactoring for dynamically typed languages

The most well-developed approach can be found in **Smalltalk Refactoring Browser**.

It relies on runtime instrumentation and existence of test suite that ensures that behaviour is preserved.

The aim of that paper was to develop refactoring tool **without** extra test suite.



The screenshot shows the Smalltalk Refactoring Browser interface. The title bar reads "IEDemoController >> oleControl1BeforeNavigate2:arg1:arg2:arg3:arg4:arg5:arg6:". The menu bar includes "Browser", "Edit", "Find", "View", "Package", "Class", "Protocol", "Method", "Tools", and "Help". The toolbar contains various icons for navigation and editing. The left pane shows a package hierarchy with "Globals" selected, containing "IEDemoController". The right pane shows the "methods" list for "IEDemoController", with "oleControl1BeforeNavigate2:" selected. The main pane displays the source code for the selected method, which is a comment and a block of Smalltalk code. The status bar at the bottom shows "Method: #oleControl1BeforeNavigate2:arg1", "Parcel: none", and "Package: c_vw7.4_image_ost8_IEDemoCon".

```
oleControl1BeforeNavigate2: arg0 arg1: arg1 arg2: arg2 arg3: arg3 arg4: arg4 arg5: arg5 arg6: arg6
" This method demonstrates using the BeforeNavigate2 event from the IE OCX to control
whether navigation to an URL is allowed.
"
| url |
url := arg1 value.
url isNil ifTrue: [ ^nil ].
(url indexOf: 'about:objectstudio@') > 0 ifTrue: [
    self modalErrorMessage: (url copyFrom: (url indexOf: '@')+1 to: url size) .
    arg6 value: 16rFFFF. "URL was handled by ObjectStudio -> Cancel navigation"
    ^self
].
```

Small example – there will be more...

Lets say, that we have class C, and field 'f' in it.
We would like to rename 'f' to 'g'...

In Java it is quite simple... For example when our refactoring tool see 'e.f', it checks type of 'e'. If it is C, it changes 'e.f' to 'e.g'... Simple...

Small example – there will be more...

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
- reference a property by specifying its name as a dynamically computed string

Major contributions of the paper

- 1) Present a **framework for specifying and implementing JavaScript refactorings**, based on a set of analysis queries on top of a pointer analysis.
- 2) Give **detailed specifications of JavaScript-specific refactorings** expressed using the framework.
- 3) **Experimentally validate** presented approach by exercising a prototype implementation of the framework and the refactorings on a set of JavaScript benchmarks.

Examples

```
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 library that defines two shapes: circles and rectangles.

```

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);

```

A client application that uses the library to draw a number of such shapes of randomly chosen sizes at random coordinates in the browser.

As a prototype-based language, JavaScript does not have built-in support for classes. Instead, they are commonly simulated using *constructor functions*.

In the library we mentioned before there are two *constructor functions*: **Circle** and **Rectangle**. They allow a programmer to create those shapes by simple **'new'** operator.

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 in this object, but the object does not itself define property **x**, the internal prototype is searched for **x** instead.

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, there is only one copy of getArea, which is shared by all rectangles.

shapes[i].getArea() (line 45) will invoke function found in internal prototype of object **shapes[i]**, but its receiver object is **shapes[i]** itself. That's why i.e. **this.width** will refer to property **width** of right object.

RENAME

- The property **this.x** (line 2) in Circle can be renamed to **xCoord**.
- This requires updating the property expression **this.x** (line 8) to **this.xCoord** as well.
- However, there is *no need* to rename the property expression **this.x** (line 14), because the properties accessed on lines 8 and 14 must reside in different objects.
- If we decide to rename **this.x** (line 14) to **this.xCoord**, then the subsequent property expression on line 21 must also be changed to **this.xCoord**.

RENAME

- Refactoring the property expression **this.drawShape** (line 6) in Circle to **this.draw** requires that the property expression **this.drawShape** (line 19) in Rectangle is refactored to **this.draw** as well
- The receivers in the expression **s.drawShape(gr)** (line 33) can be bound to a Circle or a Rectangle object, and therefore the methods have to be renamed consistently.
- Circle and Rectangle are completely *unrelated*; in particular there is no prototype relationship.

Key correctness requirement

name binding preservation — each use of a property in the refactored program should refer to the same property as in the original program.

```
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);
```

RENAME

- Applying RENAME to **this.radius** (line 4) is problematic because of the for-in loop and dynamic property expression in **dble**.
- In general, dynamic property expressions may use values computed at runtime, which would spoil any static analysis.
- In order to ensure that dynamic property expressions do not cause changes in program behavior when applying the RENAME refactoring, presented approach disallow the renaming of any property in any object on which properties may be accessed reflectively.
- Hence, in this example, it is disallowed to rename any of the properties in Circle objects.

RENAME

- 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
- Since it is now disallowed to rename any of the properties in Circle, it must be also disallowed to rename **drawShape** in Rectangle.
- The remaining properties of Rectangle, i.e., x, y, width, and height can still be renamed.

ENCAPSULATE PROPERITY

This refactoring can be used for making a field **private** and introducing new getters and setters to it. Not in JavaScript...

A commonly used technique uses local variables of constructor functions to simulate private properties.


```
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 };
```

ENCAPSULATE PROPERITY

```
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 }
```

Lets try to encapsulate **radius** field in Circle.

There is a problem! **dble** function tries to double it but it won't be found!

What is more, it will copy **drawShape** but copied function will continue to refer to original Circle!

Another short ENCAPSULATE PROPERTY example

```
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]);
```

Suppose that we want to apply ENCAPSULATE PROPERTY to the **width** property of Rectangle.

The original version of the program draws a red 100-by-200 rectangle. However, if **width** is encapsulated a red 300-by-200 rectangle is drawn instead.

EXTRACT MODULE

```
120 var geometry = (function(){
121     function Circle (x, y, r, c) {
122         this.x = x;
123         this.y = y;
124         this.radius = r;
125         this.color = c;
126         this.drawShape = function (gr) {
127             gr.fillCircle(new jsColor(this.color),
128                           new jsPoint(this.x,this.y),
129                           this.radius);
130         };
131     }
132
133     function Rectangle (x, y, w, h, c) {
134         this.x = x;
135         this.y = y;
136         this.width = w;
137         this.height = h;
138         this.color = c;
139         this.drawShape = function (gr) {
140             gr.fillRectangle(new jsColor(this.color),
141                              new jsPoint(this.x,this.y),
142                              this.width, this.height);
143         };
144     }
145     Rectangle.prototype.getArea = function() {
146         return this.width * this.height;
147     };
148
149     return {
150         Circle : Circle,
151         Rectangle : Rectangle
152     };
153 })();

154 function r(n) { return Math.round(Math.random() * n); }
155
156 function drawAll(shapes) {
157     var gr = new jsGraphics(document.getElementById("canvas"));
158     shapes.map( function(s) { s.drawShape(gr); });
159 }
160
161 var shapes = [];
162 for (var i = 0; i < 500; i++) {
163     var o = new jsColor().rgbToHex(r(255), r(255), r(255));
164     switch(r(2)) {
165         case 0:
166             shapes[i] =
167                 new geometry.Circle(r(500),r(500),r(50), o);
168             break;
169         case 1:
170             shapes[i] =
171                 new geometry.Rectangle(r(500),r(500),r(50), r(50), o);
172             alert(shapes[i].getArea());
173             break;
174     }
175 }
176 drawAll(shapes);
```

EXTRACT MODULE - problems

Observe that choosing the name **shapes** for the new module is problematic because a variable with the same name is already declared.

If the refactoring was performed anyway, the shapes “module” would be overwritten, and the constructor calls (lines 167 and 171) would cause runtime errors since the empty array shapes does not have properties Circle or Rectangle.

A Framework for Refactoring with Pointer Analysis

Basic queries

As the foundation of the framework, the paper assume a pointer analysis that defines a finite set L of object labels such that every object at runtime is represented by a label.

It is assumed that L includes labels to represent environment records.

Basic queries

For technical reasons, it is required that if an object 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.

Basic queries

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 $\ell \in L$.

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”.

Basic queries

The pointer analysis should provide the following queries:

objects For any expression e in the program, $objects(e) \subseteq L$ over-approximates the set of objects to which e may evaluate, including objects arising from **ToObject** conversion.

For a function declaration f , $objects(f)$ over-approximates the set of function objects that may result from evaluating f .

Basic queries

scope For any function definition or catch clause e , $scope(e) \subseteq L$ over-approximates the set of environment records corresponding to e at runtime. We additionally define $scope(e) := objects(e)$ for any **'with'** expression e .

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

Basic queries

props For any object label ℓ , $props(\ell) \subseteq L$
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 :

- *mayHaveProp*(ℓ , p) should hold whenever any object represented by ℓ may have a property p
- *mustHaveProp*(ℓ , p) 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).

Basic queries

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

builtin Given the name n of a built-in object, $builtin(n)$ returns the corresponding object label. The object label of the global object will be denoted as *global*.

We also define

$apply := builtin(\text{Function.prototype.apply})$

$bind := builtin(\text{Function.prototype.bind})$

$call := builtin(\text{Function.prototype.call})$

Visited and base objects

In JavaScript while evaluating i.e. '**e.x**' property **x** is looked up in object *o_1* that **e** evaluates to.

If it is not found, its prototype object *o_2* is examined... And so on, up to *o_n*. It can be found in *o_n* – than *o_n* is called a **base object** of the lookup. If it is not found – lookup returns **undefined** value.

o_1, o_2, ... ,o_n are **visited objects**.

Visited and base objects

access refer to both identifier references (like `r` on line 4) and property expressions, including both fixed-property expressions like **`s.drawShape`** and dynamic ones like **`nc[a]`** on. Identifier references and fixed-property expressions are called *named accesses*.

Visited and base objects

possiblyNamed *possiblyNamed(p)* over-approximates all accesses in the program that possibly have name p in some execution.

definitelyNamed *definitelyNamed(p)* under-approximates accesses that definitely have name p in every execution.

Visited and base objects

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

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

Visited and base objects

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

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

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

Related Accesses

We'll call two accesses $a1$ and $a2$ *directly related* if their base object may be the same and they may refer to the same property name.

The set $related(a1)$ of accesses *related* to $a1$ is computed as the smallest set R satisfying the following two closure conditions:

- $a1 \in R$;
- for every $a \in R$, if a' is an access such that a and a' are directly related, then also $a' \in R$.

Initializing functions

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.

Lets define an over-approximation of the set of possible *callees* of an invocation expression c by $callees(c) := objects(c_f)$ where c_f is the part of c containing the invoked expression.

Initializing functions

Given a function definition f , an under-approximation $initializes(f)$ of the set of objects that f initializes can be determined by ensuring the following:

f is only invoked through **new**, that is:

- No function/method call c has

$$callees(c) \cap objects(f) \neq \emptyset$$

- f is not invoked reflectively, i.e.,

$$args(apply, 0) \cap objects(f) = \emptyset,$$

and similarly for `bind` and `call`.

For any **new** expression n with

$$callees(n) \cap objects(f) \neq \emptyset$$

we have

$$callees(n) \subseteq objects(f)$$

This ensures that n definitely calls f .

Initializing functions

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
 $\mathit{initializes}(f) := \emptyset$.

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

Well-scopedness example

```
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();
```

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*.

Some more queries

- $wellscoped(f, g)$ - For a function definition node f , $wellscoped(f, g)$ holds if f is well-scoped in g .
- $intrinsic(\ell, p)$ holds whenever p is an intrinsic (i.e. **length**, **src**) property on an object labelled by ℓ .
- $reflPropAcc(\ell)$ holds whenever a property of an object labelled by ℓ may be accessed reflectively by a build-in function.

RENAME

Input A named access a and a new name y .

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

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

RENAME

Preconditions

1. x is not an intrinsic property on B :

$$\forall \ell \in B: \neg \text{intrinsic}(\ell, x)$$

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

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

3. The accesses in $\text{related}(a)$ can be renamed to y without name capture:

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

In this case, we will also say that y is *free* for $\text{related}(a)$.

4. y does not cause name capture on B , that is:

- (a) Existing accesses are not captured:

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

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

$$\forall \ell \in B: \neg \text{intrinsic}(\ell, y)$$

- (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 $B \cap \text{objects}(e) = \emptyset$.
- ii. We must have $\forall \ell \in B: \neg \text{reflPropAcc}(\ell)$.

Transformation

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

ENCAPSULATE PROPERITY

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

ENCAPSULATE PROPERITY

Preconditions

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

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.,

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

3. Every access in $\text{related}(a)$ 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 $\text{related}(a) = A_i \uplus A_g \uplus A_s$ 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. $\text{wellscoped}(f, c)$.

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:

$$\forall a \in A_i : \text{lookup}(a, x) \subseteq \{\text{global}\}$$

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

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

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

$$\forall \ell \in B : \neg \text{intrinsic}(\ell, x)$$

7. If $A_g \neq \emptyset$ then g must be free for A_g and must not cause name capture on $\text{initializes}(c)$ (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 $\text{initializes}(c)$.

ENCAPSULATE PROPERITY

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.

EXTRACT MODULE

Input Contiguous top-level statements

s_1, \dots, s_m containing a set

$P = \{p_1, \dots, p_n\}$ of identifiers to extract and
an identifier M to be used as module name.

```
s1;  
:  
:  
sm;
```

⇒

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

EXTRACT MODULE

Definitions Let S be the set of all accesses appearing in the statements s_1, \dots, 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 *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.

EXTRACT MODULE

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 $a \in A_p$:

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

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. $u \in I \vee (u \in S \wedge \neg \text{mutable}(p))$
 - ii. u is an identifier reference.
 - iii. $\text{lookup}(u, p) \subseteq \{\text{global}\}$.
3. M does not cause name capture on *global* (cf. precondition 4 of RENAME).
4. No $p \in P$ is an intrinsic on *global*:

$$\forall \ell \in B: \neg \text{intrinsic}(\ell, p)$$

Transformation

Replace s_1, \dots, s_m with the definition of module M as shown two slides before; qualify accesses in Q with M .

Implementation

Implementation – plug-in to Eclipse

- Derive a flow graph from source code.
- Create a def-use graph (it abstracts away control flow and **with** statements) from flow graph.
- Run a pointer analysis on it.
- For context sensitivity object sensitivity is used with heap specialization and a simple widening function.
- Several build-in functions are supported.
- HTML DOM is modelled with some variables like **document** initialized to point to it.

Evaluation

Evaluation

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

Evaluation

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.

50 JavaScript programs, 300 to 1700 lines of code each.

On a 3.0 GHz PC, each benchmark is analyzed in less than 4 seconds using 256 MB memory.

RENAME

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.

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.

ENCAPSULATE PROPERTY

refactoring	total applications	successful applications	rejected applications				justified
			total	imprecise preconditions	imprecise queries	imprecise analysis	
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.

The 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.

EXTRACT MODULE (1)

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.

In this experiment, for every benchmark, the code in each HTML script element was extracted into its own module. In the case of standalone benchmarks source files were chosen as the unit of modularization instead.

All failures are justified – there were problems with events handling.

EXTRACT MODULE (2)

refactoring	total applications	successful applications	rejected applications				justified
			total	imprecise preconditions	imprecise queries	imprecise analysis	
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.

In the second experiment, that was manually determined a suitable modularization for a subset of benchmarks and used our tool to perform it.

This time problems also were connected with event handling – that's why are justified.

Summary

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.

Summary

Q2: Rejections due to derived queries

The derived queries are always sufficiently precise in all experiments. For instance, ENCAPSULATE PROPERTY needs to prove well-scopedness for 28 functions, and all of them are indeed shown to be well-scoped.

Summary

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.

Summary

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.

Summary

Overall, the results of evaluation are promising.

Most attempted refactorings are performed successfully, and when the tool rejects a refactoring it mostly does so for a good reason.

Related works

To find out about related works I highly recommend reading chapter 7 of this paper. There are a lot of interesting information connected with field of refactoring.

Thinking about refactoring started in early 1990s. Since then people focused on creating automated refactoring tools...

Conclusion

- In this paper authors showed working framework for JavaScript refactoring.
- Results of testing are very promising.
- If framework refuses to refactor code, it usually makes it for a good reason.
- Authors are going to keep working on this problem. They are going to adapt their ideas to other dynamically typed languages.

Bibliography

- All of materials used in this presentation are available here:
<http://www.brics.dk/jsrefactor/index.html>
- You can find there pdf document with article (for free), another pdf with technical details about implementation and ready Eclipse plug-in.
- Thank you for your attention.