Semi-Automatic Rename Refactoring for JavaScript

Asger Feldthaus
Aarhus University
asf@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

Abstract

Modern IDEs support automated refactoring for many programming languages, but support for JavaScript is still primitive. To perform renaming, which is one of the fundamental refactorings, there is often no practical alternative to simple syntactic search-and-replace. Although more sophisticated alternatives have been developed, they are limited by whole-program assumptions and poor scalability.

We propose a technique for semi-automatic refactoring for JavaScript, with a focus on renaming. Unlike traditional refactoring algorithms, semi-automatic refactoring works by a combination of static analysis and interaction with the programmer. With this pragmatic approach, we can provide scalable and effective refactoring support for real-world code, including libraries and incomplete applications. Through a series of experiments that estimate how much manual effort our technique demands from the programmer, we show that our approach is a useful improvement compared to search-and-replace tools.

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

General Terms Languages

Keywords Refactoring, Dynamic Programming Languages, Program Analysis

1. Introduction

Refactoring is the process of transforming the source code of a program to enhance its internal structure while preserving its external behavior. Many kinds of refactoring are used in modern software development, for example, renaming of fields or methods, extracting blocks of code into separate methods, or moving code from one package to another [4]. Refactoring tools in IDEs assist the programmer in performing the necessary source code transformations and checking that the program behavior is preserved. However, the most powerful existing techniques that provide such automated support for refactoring have been developed for statically typed programming languages, such as Java and C#. As those techniques critically depend on static information about class hierarchies, packages, and types of fields and methods, they cannot easily be adapted to dynamically typed languages. In JavaScript, for example, the object properties change at runtime, and classes and packages are at best mimicked by meta-programming in libraries, which are difficult to reason about statically. The refactoring techniques that exist for dynamically typed languages, in particular JavaScript, remain primitive or impractical:

(1) One approach, introduced in the Smalltalk Refactoring Browser [9], is to use runtime instrumentation to detect the consequences of tentative program transformations. This requires extensive test suites to ensure that the program behavior is preserved, and we are not aware of this approach being used in practice for JavaScript.

(2) In previous work [3], we showed that static points-to information can be used in lieu of static types as the foundation for automated refactoring for JavaScript. Although this approach works well in many situations, we have observed some practical limitations. Pointer analysis is hard to scale to large JavaScript programs without sacrificing the necessary precision. Despite much effort, no existing pointer analysis can handle typical JavaScript programs that use libraries. Moreover, the technique only works for complete programs, which means it cannot reliably be applied to library code, except if a comprehensive test suite is available, or to application code that is under development where parts of the code have not yet been written.

(3) If we consider one of the most common kinds of refactoring—renaming an object property—the working programmer's best tool is still search-and-replace. This can be based on the plain text or the abstract syntax tree of the program source code, in either case without any consideration toward its semantics. If the programmer wishes to rename an object property, he may choose to rename all occurrences of the selected symbol in one fell swoop, or pick one-by-one the occurrences he wishes to rename. The former is prone to errors since unrelated objects may have properties with the same name, while the latter can be rather tedious if there are many occurrences.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509520

Several IDEs exist for JavaScript programming, including Eclipse JSDT, Eclipse Orion, NetBeans, Visual Studio for Web, Komodo IDE, Cloud9, and WebStorm. Among all these, we find only one—WebStorm[1]—that supports rename refactoring for JavaScript. While this tool is quite effective for renaming local variables, it seems to fall back to search-and-replace when renaming object properties. Based on brief experimentation, it appears that it always replaces all occurrences of the given object property name. WebStorm is closed source and its algorithms are not publicly documented, so we cannot verify how the tool works.

In this work, we propose a *semi-automatic* technique for renaming object properties in JavaScript programs. Renaming is among the most common kinds of refactoring [8, 12]. Making better tool support for renaming than search-and-replace requires information about how object property access operations in the program code are related to each other. Such information is also fundamental for other refactorings, for example, property encapsulation [3]. Our approach may be seen as a middle ground between fully automated refactoring and tedious one-by-one search-and-replace. When the programmer decides that some occurrence of an object property x in the program needs to be renamed to y, we use a lightweight static analysis to divide all occurrences of x into groups. Two occurrences of x are placed in the same group if the analysis infers that they are *related* in the sense that they must either both be renamed or neither be renamed. The refactoring tool then asks the programmer whether each group should be renamed entirely or left unchanged. In this way, the programmer gets one question for each group, and each question only mentions one occurrence of x. Compared to traditional search-and-replace, this approach requires less effort by the programmer since we avoid asking questions where the answer can be inferred from previous answers. Compared to the fully automated approach from our previous work, we circumvent the computationally expensive pointer analysis and the whole-program assumption.

For an incomplete program—for example, a library or an application under development—ultimately only the programmer knows whether two object property operations are related or not, given that no formal specification exists of the desired program behavior. Still, we can define an informal notion of soundness for our static analysis: if the heuristics employed by the analysis group together two occurrences of x, then it should not be the case that the programmer wants one of them to be renamed but not the other, since he only gets a question for one of them. Our static analysis therefore aims to compute a lower approximation of relatedness. The converse case, dividing the occurrences of x into too many groups, is tolerable although undesirable, since the programmer must then answer more questions to complete the refactoring. For complete programs, which have a well-defined meaning, it is reasonable to assume that two occurrences of x are related if they refer to the same object property in some execution of the program. This makes our analysis akin to alias analysis, but designed to work also on incomplete programs. The analysis is expressed as a set of rules reminiscent of a type system. We evaluate how it works on incomplete programs by measuring the stability of the analysis behavior when source code is added or removed from the codebase. The analysis is theoretically unsound, but our experiments indicate that it is sound in practice.

Some refactoring tools follow a tradition of performing safety checks to ensure that the program behavior will remain unchanged when performing refactorings. If a precondition is violated, such tools will abort or issue a warning to the programmer. For example, a common precondition for renaming a field in a Java program is that it does not cause a name clash. These safety checks are usually not sound in practice. Java refactoring tools typically ignore the possibility of reflection, while refactoring for JavaScript may give some leeway regarding use of the `eval` function. In the opposite end of the spectrum, other refactoring tools assume that programmer knows what he is doing, so no safety checking is performed. We follow the latter approach. The foremost rationale for this decision is that proper precondition checking requires static analysis that scales to large real-world codebases, has high precision, and works on incomplete programs, which is beyond the capabilities of existing JavaScript analysis techniques. We argue that this decision is not as detrimental to safety for semi-automatic refactoring as it would be for a fully automatic tool: the programmer will observe a number of proposed changes in the code while interacting with the tool, so he is more likely to detect problematic situations.

Reflective property access is common in JavaScript. Such expressions access an object property whose name is computed at runtime. Reflective property accesses cannot be renamed directly, since the source code does not mention the property name explicitly, and it lies beyond the scope of our proposed algorithm to update the relevant string operations that compute the property name. Our algorithm will only rename identifier tokens in the source code, and the user is expected to know this. Apart from this assumption, the internals of our proposed algorithm do not concern the programmer when performing refactorings.

Our contributions can then be summarized as follows:

- We present a novel approach to semi-automatic refactoring, based on an algorithm for finding groups of related identifier tokens. A key component of this algorithm is a type inference technique reminiscent of alias analysis.
- We present experiments that evaluate the practical usefulness of the approach. The results show that (1) it asks significantly fewer questions than search-and-replace, (2) it scales to large real-world codebases, (3) it does not require the whole program to be useful, and (4) although unsound in theory, the unsoundness rarely manifests in practice.

[1] `http://www.jetbrains.com/webstorm/`

In Section 2 we motivate our technique using a real-world code snippet. In Section 3 we present the algorithm and demonstrate it on a few examples, and in Section 4 we present research questions and results of the experimental evaluation. Related work is discussed in Section 5, and our conclusions and suggestions for further work are given in Section 6.

2. Motivating Example

Renaming a local variable is trivial in most cases, as the scoping rules in JavaScript let us find all tokens that refer to the variable. However, renaming object properties is non-trivial. In this section, we will demonstrate some of the challenges involved in renaming object properties.

Consider the code fragment in Figure 1, taken from the popular jQuery library.[2] We shall intentionally withhold information about the rest of jQuery for now, and see what we can learn about the code without providing any context. The code fragment contains a top-level statement that creates a new object with six properties, three of which are initialized to function expressions. The new object is then stored in the `prototype` property of the `jQuery.Event` object. The `prototype` property has a special meaning in JavaScript: informally, objects created using a constructor call `new jQuery.Event` will inherit the properties of `jQuery.Event.prototype`.

We will now discuss how an editor with our refactoring technique reacts to various renaming request from a programmer, assuming the code in Figure 1 is the only code open in the editor. Although refactoring a fragment of jQuery is an unusual scenario, the code in Figure 1 is representative of some of the issues we generally encounter when refactoring incomplete programs.

Consider the three `preventDefault` tokens on lines 2, 6, and 7. Suppose the programmer has decided to rename the token on line 2. It is not immediately evident whether the tokens on line 6 and 7 should be renamed as well. The IDE therefore asks the programmer whether the token on line 6 should be renamed also. However, regardless of the answer, the IDE will not ask a second time whether the token on line 7 should be renamed. Clearly, if one of these tokens were to be renamed, the other must also be renamed. We call such tokens *related*. In this particular case, our technique will determine that they are related because they both occur as the name of a property accessed on the `e` variable. In short, we say they *access* the same property on `e`.

The `originalEvent` tokens on lines 4 and 14 access a property on `this` but in different functions. Technically, any function can be invoked with an arbitrary `this` argument, and thus the two uses of `this` could refer to very different objects that just happen to both have a property called `originalEvent`. In practice, though, such a scenario is unlikely. Due to the function call semantics of JavaScript, the object on which a function is stored is often the `this` argument or a prototype thereof. Hence our analysis will determine that the two `originalEvent` tokens are related. By similar reasoning, the `isDefaultPrevented` tokens on lines 3 and 26 are also considered related.

```
 1 jQuery.Event.prototype = {
 2    preventDefault: function() {
 3       this.isDefaultPrevented = returnTrue;
 4       var e = this.originalEvent;
 5       if ( !e ) { return; }
 6       if ( e.preventDefault ) {
 7          e.preventDefault();
 8       } else {
 9          e.returnValue = false;
10       }
11    },
12    stopPropagation: function() {
13       this.isPropagationStopped = returnTrue;
14       var e = this.originalEvent;
15       if ( !e ) { return; }
16       if ( e.stopPropagation ) {
17          e.stopPropagation();
18       }
19       e.cancelBubble = true;
20    },
21    stopImmediatePropagation: function() {
22       this.isImmediatePropagationStopped
23          = returnTrue;
24       this.stopPropagation();
25    },
26    isDefaultPrevented: returnFalse,
27    isPropagationStopped: returnFalse,
28    isImmediatePropagationStopped: returnFalse
29 };
```

Figure 1. A fragment of jQuery (any version after 1.5.1).

Consider now `stopPropagation` on lines 12, 16, 17, and 24. The tokens on lines 16 and 17 are considered related, as are those on lines 12 and 24. But it is not evident whether these two groups are also related to each other. Two of them access a property on a jQuery event object or its prototype, while the other two access a property on `e`, which is an alias for `this.originalEvent`. There is no reliable indicator as to whether these refer to the same kind of object. The call on line 17 could be a recursive call to the function defined on line 12, or it could be a call to a lower-level event object whose method happens to have the same name. Our analysis considers the two groups of tokens unrelated from each other, and to rename the four tokens the programmer must thus answer two questions, one for each group.

In jQuery, the `originalEvent` property happens to refer to a native event object, created by the browser. As such, the `stopPropagation` tokens on lines 16 and 17 refer to a property on a native event object, while the tokens on lines 12 and 24 refer to a property on jQuery's own event objects, which function as wrappers around the native event objects.

[2] `http://jquery.com/`

Indeed, it is possible to rename the `stopPropagation` property on jQuery's event objects by updating the tokens on lines 12 and 24 and one other token elsewhere in jQuery (outside the code shown in Figure 1), while leaving the tokens on lines 16 and 17 unaltered. The result is a refactored version of the jQuery library that remains compatible with most jQuery applications. Only applications that explicitly invoke `stopPropagation` on a jQuery event object are incompatible with the new version. In this way, the programmer remains responsible for ensuring backward compatibility in case a renaming interferes with the public API of the library, but our technique makes it easier for the programmer to perform the desired refactorings of the library code.

Moreover, consider what would happen if we were to update *every* `stopPropagation` token in the previous example, that is, including those on lines 16 and 17. The call on line 17 would then no longer work as intended, because the browser would still use the name `stopPropagation` when creating native event objects, regardless of any source code transformation we have performed. As discussed previously, our technique does not perform safety checking, so we trust that the programmer does not attempt to perform such refactorings. This allows our technique to work without modeling the browser API.

A final concern when renaming a property is that all references to the property must be updated consistently. For instance, suppose we renamed *only* the `stopPropagation` token on line 12. Clearly, the call on line 24 would then fail. In this particular case, our technique groups these tokens together, and it is thus impossible for the programmer to perform the erroneous renaming using our refactoring tool. But this is not always the case; in general, the consistency of a renaming depends on the programmer answering correctly to the questions posed by the refactoring tool, which is no different than ordinary search-and-replace.

While our tool does not actively prevent inconsistent renamings, it should be able to carry out any consistent renaming that can be done by renaming of identifier tokens, assuming the programmer answers its questions correctly. To demonstrate the importance of this criterion, suppose our technique incorrectly determined that all four `stopPropagation` tokens in Figure 1 were related. In this case, the previously mentioned refactoring of jQuery would not have been possible, because it involves renaming only a subset of these tokens. Indeed, if the programmer decided to rename the token on line 12, he would not get the opportunity to tell the tool not to rename those on lines 16 and 17, leading to a different refactoring that was intended. When working with a large codebase, the programmer might not immediately notice that his renaming did not go as intended, thereby introducing a bug in his code. Such behavior is therefore highly undesirable, and while our technique is not impervious to this type of failure, our evaluation shows that it is unlikely to occur in practice.

This example demonstrates that we need a static analysis that is able to approximate how object property tokens are related in JavaScript programs. This analysis must be sound in practice to avoid undesired refactorings, it must be sufficiently precise to enable a significant reduction of programmer effort compared to traditional search-and-replace, and it must be scalable and fast to be usable during programming. In addition, we want the analysis to work robustly on incomplete programs, such as libraries or applications under development. As we discuss in Section 5, no existing static analysis analysis satisfies these requirements, which motivates the design of our new analysis described in the following section.

3. Finding Related Identifier Tokens

Our proposed refactoring algorithm depends on a reliable method for finding related tokens in the program code. We define an ad-hoc type inference system, based on how objects appear to be used. Unlike most type systems, ours does not perform type *checking*, that is, there is no such thing as a static type error; all programs can be typed. A program typing is an equivalence relation between expressions, denoting which expressions have the same type. Given two occurrences of expressions, $e_1.f$ and $e_2.f$, we can subsequently say that the two f tokens are *related* if e_1 and e_2 have the same type. The type inference is considered *sound* if it is never the case that two tokens are found to be related while the programmer only intends to rename one and not the other.

The type inference is performed in two phases, denoted *basic type inference* and *receiver type inference*. Each phase traverses the abstract syntax tree once, generating a set of constraints. At the end of each phase, these constraints are saturated under a collection of closure rules using an augmented union-find data structure.

Our type analysis is reminiscent of alias analysis, in particular, the Steensgaard style [11], however, with important differences. Alias analysis determines which expressions in a program are *aliased*, that is, refer to the same object. The most common kind is *may*-alias analysis, which finds pairs of expressions that may, in *some* execution, evaluate to the same object. Given two expressions $e_1\texttt{.}f$ and $e_2\texttt{.}f$, one might consider the two f tokens to be related if e_1 and e_2 are may-aliased. However, may-alias analysis is conservative in the direction opposite of what we want: it may report some expression pairs as may-aliased even if they can, in fact, never be aliased. When this happens, unrelated tokens may be classified as related, which, as previously mentioned, is highly undesirable. Instead, one may consider *must*-alias analysis, which finds pairs of expressions that are aliases in *every* execution. This will result in a sound construction of related tokens, however, the resulting precision will be poor, even for a perfectly precise must-alias analysis. As an example, consider a composite expression `x.f + y.f` where `x` and `y` are aliases in some program executions but not in others. In this case, using a must-alias analysis will result in the two `f` tokens to be considered non-related. However, we would like to treat them as related, because there exists an execution where they refer to the same object property. This means that we wish to design a *must-sometimes*-alias analysis that finds pairs of expressions that *must* be aliases in *some* execution. We are not aware of existing analyses of that kind.

Moreover, our analysis must account for object prototypes, which JavaScript uses for mimicking inheritance, and certain common programming patterns, such as, prototype methods and object extension functions. For reasons explained in the following sections, these patterns indicate relatedness without involving aliasing, so we refer to our analysis as a *type inference* rather than an alias analysis.

3.1 Constraint Syntax

The constraints generated are of form $t_1 \equiv t_2$ where t_1, t_2 are *terms*. A term may either be an expression e, or a compound $e \diamond f$ representing the f property on expression e. The $\equiv$ relation is called the *same-type* relation. We say that two terms, t_1 and t_2, have the same type when $t_1 \equiv t_2$. As an example, $\texttt{x} \diamond \texttt{f} \equiv \texttt{y}$ means that the variable `x` points to an object with an `f` property that has the same type as the variable `y`.

The schema below summarizes the syntax and our naming conventions:

$o, v, e \in$ expressions
$g \in$ function bodies
$f \in$ identifiers
$t \in$ terms
$t ::= e \mid e \diamond f$
$\equiv \subseteq$ terms $\times$ terms

We reserve the meta-variable o for object literals, v for program variables, and e for any type of expression. The notion of an *expression* is used quite liberally, but the meaning should be clear from the context: We use it primarily when referring to an occurrence of a JavaScript expression in the abstract syntax tree; thus, identical expressions at different positions in the program code are considered distinct. In a slight abuse of terminology, we also refer to variable declarations, such as a `var` statement or the token of a parameter name, as expressions. Finally, we include the following pseudo-expressions:

glob: expression representing the global object

ret(g): expression representing return value of function g

this(g): expression representing `this` inside function g

Unlike expressions, the notion of an *identifier* does *not* denote a specific occurrence in the abstract syntax tree; two identifiers are indistinguishable if they consist of the same string of characters. We will use the notion of *tokens* when different occurrences of the same identifier should be considered distinct. The artificial identifier `[array]` refers to array entries.

statement or expression e		constraints
variable	v	$e \equiv$ *decl*(v)
property	e_1`.`f	$e \equiv e_1 \diamond f$
dyn. prop.	e_1`[`e_2`]`	see text
*assignment	e_1`=`e_2	$e \equiv e_1 \equiv e_2$
*conditional	e_1 `?` e_2 `:` e_3	$e \equiv e_2 \equiv e_3$
*logical or	e_1 `||` e_2	$e \equiv e_1 \equiv e_2$
logical and	e_1 `&&` e_2	$e \equiv e_2$
this	`this`	$e \equiv$ *this*(*fun*(e))
return	`return` e_1	$e_1 \equiv$ *ret*(*fun*(e))
call	e_1`(`$\vec{e}_2$`)`	see text
new-call	`new` e_1`(`$\vec{e}_2$`)`	see text
function	`function` f`(`$\vec{v}$`){ ... }`	see text
array literal	`[` e_1`,`e_2`, ... ]`	$e \diamond$ `[array]` $\equiv e_i$
object literal	`{ ... }`	see below

member of object literal o		constraints
initializer	f`:`e_1	$o \diamond f \equiv e_1$
getter	`get` f`(){`$_g$ `... }`	$o \diamond f \equiv$ *ret*(g) $o \equiv$ *this*(g)
setter	`set` f`(`v`){`$_g$ `... }`	$o \diamond f \equiv v$ $o \equiv$ *this*(g)

Figure 2. Constraints for basic type inference. A star * indicates that an exception to the rule is described in the text.

3.2 Saturation

At the end of each phase, we saturate the $\equiv$ relation until it satisfies a collection of closure rules. In particular, we ensure that $\equiv$ is an equivalence relation:

$$\frac{}{t \equiv t}\ \textit{(refl)} \qquad \frac{t_1 \equiv t_2}{t_2 \equiv t_1}\ \textit{(sym)} \qquad \frac{t_1 \equiv t_2 \quad t_2 \equiv t_3}{t_1 \equiv t_3}\ \textit{(trans)}$$

and that $\equiv$ moreover satisfies the following rule:

$$\frac{e_1 \equiv e_2}{e_1 \diamond f \equiv e_2 \diamond f}\ \textit{(prty)}$$

Informally, the *prty* rule states that same-typed expressions have same-typed properties. Examples in the next section demonstrate the consequence of the saturation rules. We present an efficient algorithm for performing the saturation in Section 3.5.

3.3 Basic Type Inference

In the first phase we traverse the abstract syntax tree and for each expression generate constraints according to Figure 2. We use the notation $e_1 \equiv e_2 \equiv e_3$ as shorthand for the two constraints $e_1 \equiv e_2$ and $e_2 \equiv e_3$. We also introduce the following two auxiliary definitions:

decl(v): the declaration of v, or *glob* $\diamond\, v$ if v is global

fun(e): the innermost function containing e

We will now discuss each rule in detail.

Variable For an expression e that is a variable v, we add the constraint that v should have the same type as its declaration: $e \equiv \mathit{decl}(v)$. By transitivity, all uses of the variable will thereby have the same type.

Example The three uses of `v` below will have same type as the declaration of `v` on line 30 due to the variable rule. After saturation (Section 3.2), they will then all have the same type due to transitivity, and thus the two `x` tokens will ultimately be considered related:

```
30 function f(v) {
31     v.x = v.y;
32     return v.x;
33 }
```

Property When a property expression e of form e_1`.`f is encountered, we add the constraint $e \equiv e_1 \diamond f$. Due to the *prty* rule, if f is accessed on a similarly typed expression, the two accesses will then be given the same type.

Example As per the previous example, the three uses of `v` below are same-typed. We use subscripts to name tokens in the example code:

```
34 function f(v) {
35   v₁.x.y₃ = v.y;
36   return v₂.x.y₄;
37 }
```

Because $v_1 \equiv v_2$, the *prty* rule yields the typing $v_1 \diamond x \equiv v_2 \diamond x$. Since $v_1.x \equiv v_1 \diamond x$ and $v_2.x \equiv v_2 \diamond x$ were generated while traversing the abstract syntax tree, we get by transitivity that the two `v.x` expressions are same-typed. Thus, y_3 and y_4 will ultimately be considered related.

Dynamic property An expression of form e_1`[`e_2`]` performs an array access or a reflective property access on e_1 (technically, they are the same in JavaScript). The name of the property being accessed depends on the value of e_2 at runtime. If e_2 is a string constant "f" we treat the expression as $e_1.f$; in all other cases, we ignore the expression.

Assignment For an expression e of form $e_1 = e_2$, we add the constraints $e \equiv e_1$ and $e \equiv e_2$. There is an exception to this rule, however. Consider this chain assignment:

```
38 x = y = null;
```

Such a statement is often employed as a compact way to clear the value of several variables, but is generally not a good indicator of the variables `x` and `y` having the same type. Indeed, no object could be assigned to both `x` and `y` by executing the statement. We classify certain expressions as *primitive* when they definitely cannot evaluate to an object. Null expressions are primitive, and an assignment expression is primitive if its right-hand side is primitive. If the right-hand side of an assignment is primitive, then we disregard the above rule for assignments and generate no constraints. No constraints are generated for compound assignment operators, such as, +=.

Conditional An expression e of form e_1 `?` e_2 `:` e_3 evaluates e_1 and then evaluates and returns the value of either e_2 or e_3, depending on whether e_1 was true or false. We therefore add the constraints $e \equiv e_2$ and $e \equiv e_3$.

There is an exception to the above rule, however, since programmers occasionally use the `?:` operator in situations where an if-statement would have been appropriate. The following two statements are semantically equivalent:

```
39 if (b) x = y else z = w;
40 b ? (x = y) : (z = w);
```

In the latter case, the result of the `?:` expression is immediately discarded. We say that such expressions occur in *void context*. When a `?:` expression occurs in void context, we disregard the rule above and generate no constraints. Otherwise, `x` and `y` would have been considered same-typed with `z` and `w` after saturation due to transitivity.

Logical or An expression e of form e_1 `||` e_2 will at runtime evaluate e_1, and then if e_1 is false, it will evaluate e_2 and return its result. If e_1 is true, the result of e_1 is returned. Although the operator is called *logical or*, its result need not be a boolean. Objects are considered to be true when coerced to a boolean value. Hence, an object from either e_1 or e_2 may be returned; we therefore add the constraints $e \equiv e_1$ and $e \equiv e_2$. As for the `?:` operator, we disregard this rule when e occurs in void context.

Logical and An expression e of form e_1 && e_2 will at runtime evaluate e_1. If e_1 is true, it will then evaluate e_2 and return the result of e_2, and otherwise it will return the value of e_1. Since objects cannot be false when coerced to a boolean, only objects from e_2 may be returned. Thus, we add the constraint $e \equiv e_2$. The void context exception could be applied to this rule as well, but in this case it makes no difference, since $e \equiv e_1$ is not generated either way.

This For an expression e of form `this` we add the constraint that e should have the same type as the `this` argument in the enclosing function: $e \equiv \mathit{this}(\mathit{fun}(e))$. Thus, all uses of `this` in a given function will be given the same type.

Return For a return statement e of form `return` e_1, we add the constraint $e_1 \equiv \mathit{ret}(\mathit{fun}(e))$. This ensures that all returned expressions will have the same type.

Example In the function below, `a` and `b` will be same-typed because they are both returned within the same function. Thus the two `x` tokens will ultimately be considered related:

```
41 function minX(a,b) {
42   if (a.x < b.x) return a;
43   else return b;
44 }
```

Call and new Most function calls are ignored by our algorithm. Precise inference of function types is complicated for a language such as JavaScript. If a call graph were available, we could connect arguments to parameters and return values to results, but if done in a context-insensitive manner, the transitivity of the $\equiv$ relation would in practice declare too many expressions as same-typed, which, as previously discussed, is highly undesirable. If done context-sensitively, scalability would be jeopardized. In our setting, the conservative action is to exclude constraints rather than include them, and as such, ignoring function calls can be tolerated.

One particular type of function call is easily handled, however. JavaScript programs often use *one-shot closures* to obtain encapsulation:

```
45 (function(self) {
46     var x; // ‘x‘ not visible in outer scope
47     /* ... */
48 })(this);
```

A function call e of form e_0`(`e_1`,`e_2`, ...)` in which e_0 is an anonymous function expression is easily handled for two reasons: (a) the called function is known, and (b) no other call can invoke that function. For this type of call, we add the constraint $e_i \equiv v_i$ for each corresponding argument e_i and parameter v_i. Likewise, we add the constraint $\textit{ret}(e_0) \equiv e$. If the call was made with the `new` keyword, we further add the constraint $\textit{this}(e_0) \equiv e$, and otherwise $\textit{this}(e_0) \equiv \textit{glob}$ since the global object is passed as `this` argument in that case.

Example One-shot closures are often used together with for-loops as in the following example:

```
49 for (var i=0; i<10; i++) {
50     var panel = panels[i];
51     var handler = (function(panel) {
52         return function() {
53           panel.activated(true);
54         }
55     })(panel);
56     panel.button.addEvent("click", handler);
57     panel.activated(false);
58 }
```

If the one-shot closure was not used, all ten event handlers would refer to the same `panel` variable, so the ith event handler would invoke `activated` on the *last* panel, rather than the ith panel. By handling one-shot closures, our analysis finds that the two uses of `panel` on lines 53 and 57 have the same type, despite referring to different variables. Thus, the two `activated` tokens are ultimately considered related.

Function Functions are first-class objects in JavaScript, and thus may themselves have properties. One property of particular interest is the `prototype` property of a function object. When a function is invoked using the `new` keyword, a newly created object is passed as the `this` argument. This object has its internal prototype link initialized to the object pointed to by the `prototype` property of the function object, effectively inheriting the properties of this object. For any function expression e with body g, we therefore add the constraint $e \diamond$ `prototype` $\equiv \textit{this}(g)$. For named functions, we similarly add the constraint $v \diamond$ `prototype` $\equiv \textit{this}(g)$ where v is the function name declaration.

Using the `prototype` property for a purpose other than inheritance is highly unusual, even for functions that are never invoked using `new`. Thus, the constraint will typically have no impact for functions that are not intended to be invoked using `new`.

Example In the code below, a string builder function is defined, and two functions are placed on its prototype object. Due to the above rule, `this` on line 60 will have the same type as `StringBuilder.prototype` on line 62, and the two `clear` tokens will thus be considered related:

```
59 function StringBuilder() {
60     this.clear();
61 }
62 StringBuilder.prototype.clear = function() {
63     this.array = [];
64 };
65 StringBuilder.prototype.append = function(x) {
66     this.array.push(x);
67 };
68 StringBuilder.prototype.toString = function() {
69     return this.array.join("");
70 };
```

During the second phase of the algorithm, which we describe in Section 3.4, the three uses of `this` on lines 63, 66 and 69 will also get the same type, and the three `array` tokens will thus also become related.

Array literal An array literal e of form `[`e_1`,`e_2`, ...]` creates a new array object, initialized with the value e_i in its ith entry. We assume such array objects are intended to be *homogeneous*, and add the constraint $e \diamond$ `[array]` $\equiv e_i$ for each i. A homogeneous array is an array for which all elements have the same type. Not all arrays are intended to be homogeneous, but we found that those created using array literals typically are.

Note that the artificial `[array]` property is not referenced by any of the other rules. In particular, there is no rule that handles array access expressions, since such expressions are hard to distinguish from reflective property accesses, and the array being accessed might not be homogeneous.

Example Array literals are often used to write out constant tables in JavaScript source code, as in the below snippet taken from Mozilla's JavaScript PDF reader:

```
71 var QeTable = [
72   {qe: 0x5601, nmps: 1, nlps: 1, switchFlag: 1}
73  ,{qe: 0x3401, nmps: 2, nlps: 6, switchFlag: 0}
74  ,{qe: 0x1801, nmps: 3, nlps: 9, switchFlag: 0}
75   /* ... */
76 ];
```

Since each member of the array is assigned the same type, each `qe` token will be considered related, and likewise for the three other property names.

Object literal An object literal o is an expression of form `{...}` containing zero or more object literal *members*. Such an expression creates a new object, with the object literal members denoting the initial values of its properties. There are three types of members: initializers, getters, and setters.

An initializer is of form f `:` e_1 and assigns the value of e_1 to the f property of the new object. For each such initializer, we add the constraint $o \diamond f \equiv e_1$.

A getter is of form `get` f`(){...}`. This assigns a *getter* function for the f property on the new object. Whenever the f property is read from the object, the getter is invoked, and the value it returns is seen as the value of the f property. For each getter, we add the constraint $o \diamond f \equiv \mathit{ret}(g)$, where g denotes the getter function's body. We also add $o \equiv \mathit{this}(g)$, since the object is passed as `this` argument when the getter is invoked.

A setter is of form `set` f`(`v`){...}`. This assigns a *setter* function for the f property. Whenever the f property is assigned to on the object, the setter is invoked with the new value passed as argument. We therefore add the constraints $o \diamond f \equiv v$ and $o \equiv \mathit{this}(g)$, where g denotes the setter function's body.

Getter and setters were introduced in ECMAScript 5 [2] and are now used in many applications; other recently introduced language features are ignored by our analysis.

3.4 Receiver Type Inference

In the second phase, we classify certain functions as *methods* and add additional type constraints accordingly. However, reliably classifying functions as methods requires knowledge of *namespace objects*, as we shall discuss shortly. We need the same-type relation inferred during the previous phase to detect such namespace objects, hence the division into two phases. We will now motivate the informal concepts of methods as namespaces, and then discuss the type inference algorithm for this phase.

JavaScript has no first-class concept of methods or constructors, only functions and invocation conventions that let programmers mimic these features. Function calls of form e`.`f`(...)` will receive the value of e as the `this` argument, while calls of form `new` e`(...)` and `new` e`.`f`(...)` receive a newly created object as the `this` argument. Although programmers may mix and match these invocation types for any one function, most functions are in practice designed to be called exclusively one way or the other. Functions designed to be called with `new` are referred to as constructors. Likewise, a non-constructor function stored in a property on some object is sometimes referred to as a *method* of that object.

JavaScript also has no built-in support for namespaces, packages, or modules. This has led to a tradition of using objects to mimic namespaces. When a function is stored in some property of a namespace object, it may be wrong to consider it a method on the namespace, as it could just as well be a constructor.

```
77 obj.onmouseover = function() {
78     this.active = true;
79 };
80 obj.onmouseout = function() {
81     this.active = false;
82 };
```

(a)

```
83 util.Rect = function(x1,y1,w,h) {
84     this.x = x1;
85     this.y = y1;
86     this.width = w;
87     this.height = h;
88 };
89 util.Vector2 = function(x1,y1) {
90     this.x = x1;
91     this.y = y1;
92 };
93 /* ... */
94 new util.Vector2(0,10);
```

(b)

Figure 3. Functions stored as (a) methods on an object, and (b) constructors in a namespace.

Figure 3 demonstrates examples of methods and constructors stored on objects. To motivate the need to distinguish these cases, suppose the programmer has decided to rename the `active` token on line 78. For this type of code, it is generally safe to assume that the `active` token on line 81 is related to the one on line 78.

However, if instead the programmer had decided to rename the `x` token on line 84, it is not generally safe to assume that the `x` token on line 90 should be renamed as well. The key difference is that `Rect` and `Vector2` are constructors, and thus their `this` arguments are not related to the object on which the function is stored.

Unfortunately, the two cases are structurally quite similar. Given a statement of form e`.`f`=function(...){...}`, we cannot immediately tell if the function is a method or a constructor.

In this phase, we exploit two indicators of a function being a constructor. The most direct indicator is that it is invoked using the `new` keyword. The second indicator is that the source code mentions its `prototype` property. As previously mentioned, this property has a special meaning when used on functions, and is typically only used for constructor functions.

Generally detecting how a function is called is a hard problem involving call graph construction, but our situation is much simpler than that. We are primarily interested in functions that are accessed through namespace objects, which we can easily recognize statically using the following heuristic.

Namespace detection After the constraints from the prior phase have been generated, we saturate the $\equiv$ relation under the closure rules in Section 3.2. Then, for any expression of form `new` e`.`f`(...)` or e`.`f`.prototype` we mark the type of e as a *namespace* type. Any expression with same type as e will be considered to be a namespace. Namespaces are not considered to have *any* methods.

Using the above heuristic, the `Rect` function in Figure 3 will not be considered a method on the `util` object, because `util` has been marked as a namespace by the `new`-call on line 94.

There is an inherent whole-program assumption in this heuristic. If the programmer has created a namespace with two functions intended to be used as constructors, but is not yet using any of them, the heuristic may fail to detect the namespace object. For example, if the `new` call on line 94 were not yet written, the two `x` tokens in Figure 3(b) would be treated as being related. The practical implications of this are discussed in the evaluation section.

Method definition Once namespaces have been identified, we look for *method definitions*. For an expression of form e_1`.`f = e_2 where e_1 was not marked as a namespace and e_2 is a function expression, we say e_2 is a *method definition* with e_1 as its *host* expression. Likewise, for an initializer $f : e_2$ inside an object literal o, we say e_2 is a method definition with o as its host expression. For any method definition with body g and host expression e, we add the constraint $e \equiv \mathit{this}(g)$.

Example (prototype method) In the code below, the `baz` function is considered a method on `Foo.prototype`, hence its `this` argument will get the same type as `Foo.prototype`. Since `this` inside the `Foo` function also has the same type as `Foo.prototype` by the function rule from the prior phase, the two uses of `this` have the same type. Hence, the two uses of `x` will ultimately be considered related.

```
 95 function Foo(x1) {
 96    this.x = x1;
 97 }
 98 Foo.prototype.baz = function() {
 99    alert(this.x);
100 };
101 new Foo(5).baz(); // alerts "5"
```

Example (extend function) The code below uses the `extend` function commonly found in third-party libraries, which copies all properties from one object to another. The host object of the `baz` method is thus a temporary object that exists only briefly until its properties have been copied onto the `Foo` object. Even though this temporary object is never actually passed as `this` to the `baz` method on line 105, the method definition constraint ensures that the object literal has the same type as `this`, which has the desired effect: the two uses of `x` will be considered related.

```
102 var Foo = {};
103 Object.extend(Foo, {
104    x: 5,
105    baz: function() {
106       alert(this.x);
107    }
108 });
109 Foo.baz(); // alerts "5"
```

This example, as well as the following one, also demonstrates why our analysis is technically not an alias analysis: the `this` expression on line 106 is not an alias of the object literal, so if we used a precise alias analysis instead of our type inference, the two uses of `x` would not be considered related.

Example (class system) The code below uses a popular class system provided by the prototype.js library[3] to simulate the creation of a class. As with the `extend` function, the host object for the two methods is in fact not passed as `this` to either method, but again, receiver type inference has the desired effect: the two uses of `x` become related.

```
110 var Foo = Class.create({
111    initialize : function(x1) {
112       this.x = x1;
113    },
114    baz : function() {
115       alert(this.x);
116    }
117 });
118 new Foo(5).baz(); // alerts "5"
```

3.5 Saturation Algorithm

At the end of each phase, the $\equiv$ relation is saturated until it satisfies the closure rules in Section 3.2. Since the result is an equivalence relation, it can be represented efficiently using a union-find data structure [1]. In the following, we assume the reader is familiar with union-find and associated terminology.

The only rule not immediately satisfied by virtue of the traditional union-find algorithm is the *prty* rule. We augment the union-find data structure such that each node has a prty field, in addition to the standard parent pointer and rank fields. The prty field holds a map from strings (property names) to nodes of the union-find data structure. Initially, one union-find node is created for each program expression, including the pseudo-expressions defined in Section 3.1. We informally refer to these nodes as *types*, since two expressions have the same type exactly if their nodes have a common representative. The type of a term $e \diamond f$ is the node pointed to by the f entry in the prty map of the representative of e. If no such node exists, then $e \diamond f$ is not same-typed with any other term.

[3] `http://prototypejs.org/`

When two nodes n_1, n_2 are unified, such that n_1 becomes the new root, their prty maps are merged by the following procedure. For property names present only in n_2's prty map, the corresponding entry is copied over to n_1's prty map. For property names present in both maps, the nodes they refer to are recorded in a worklist of node-pairs that should be unified. As a simple and effective optimization to this procedure, we initially swap n_1 and n_2's prty pointers if n_2's prty map is bigger than n_1's, so fewer entries need to be copied. At the end of each phase, node pairs are removed from the worklist and unified until the worklist is empty.

We also store an isNamespace boolean on each node, for marking nodes as namespaces during the second phase. The namespace detection can be done while traversing the abstract syntax tree in the first phase.

A pseudo-code implementation of the augmented union-find data structure is given in Figure 4. The code does not include the generation of constraints during traversal of the abstract syntax tree.

When a constraint of form $e_1 \equiv e_2$ is discovered, we invoke the unify method with the corresponding nodes. For a constraint of form $e_1 \diamond f \equiv e_2$, we invoke unifyPrty instead. The constraints need not be stored explicitly; invoking the corresponding method on the Unifier instance is sufficient. At the end of each phase, we invoke the complete method to ensure that the *prty* closure rule is satisfied.

4. Evaluation

We implemented a renaming plugin[4] for Eclipse, based on the algorithm described in the previous section. We use the same underlying implementation for this evaluation.

Ideally, the primary metric of usefulness of a refactoring tool is its impact on programmer productivity. This is unfortunately hard to define and difficult to measure directly, so we base our evaluation on the following more tangible metrics, which we consider good indicators of usefulness:

Manual Effort: How many questions must the programmer answer to complete a renaming? Since the programmer need only consider a single token per question, a question issued by our tool is no more difficult than the corresponding search-and-replace question.

Soundness: If given correct answers by the programmer, how likely is it that a renaming is ultimately inconsistent? In other words, is the analysis sound in practice?

Delay: How long must the programmer sit idle while waiting for the tool to finish a computation?

Whole-Program: Does the tool apply to library code, without having application code available? Does it apply to incomplete application code, such as, code under development or applications without libraries?

[4] `http://www.brics.dk/jsrefactor/plugin.html`

```
class UnifyNode:
    field parent = this
    field rank = 0
    field prty = <empty map>
    field isNamespace = false

    def rep():
        if parent != this:
            parent = parent.rep()
        return parent

class Unifier:
    field queue = <empty queue>

    def unify(x,y):
        x = x.rep()
        y = y.rep()
        if x == y:
            return
        if x.rank < y.rank:
            swap x, y
        else if x.rank == y.rank:
            x.rank += 1
        y.parent = x
        x.isNamespace |= y.isNamespace
        if x.prty.size < y.prty.size:
            swap x.prty, y.prty
        for k,v in y.prty:
            if k in x.prty:
                unifyLater(x.prty[k], v)
            else:
                x.prty[k] = v
        y.prty = null

    def unifyPrty(x,k,y):
        x = x.rep()
        if k in x.prty:
            unify(x.prty[k], y)
        else:
            x.prty[k] = y

    def unifyLater(x,y):
        queue.add(x,y)

    def complete():
        while queue is not empty:
          (x,y) = queue.pop()
          unify(x,y)
```

Figure 4. Python-like pseudo-code implementation of the augmented union-find data structure.

We collected 24 JavaScript applications for use as benchmarks. Third-party libraries are used in 19 of the applications, constituting a total of 9 distinct libraries (some libraries are used by more than one application) that we also include as benchmarks. Of these 24 applications, 10 were taken from the 10k Event Apart Challenge,[5] 10 were taken from Chrome Experiments,[6] and 4 were found at GitHub.[7] When selecting benchmarks, we aimed for diversity in complexity, functionality, and use of libraries. The benchmark collection is available online.[8]

To evaluate our technique along the above metrics, we perform a series of experiments on these applications. The benchmarks and experimental results are shown in Table 1. In the following, we describe which experiments were used to evaluate the various metrics, and the meaning of the columns in Table 1.

4.1 Manual Effort

For both our technique and search-and-replace, the total number of questions asked during a refactoring is independent of whether the programmer answers yes or no to each question. It is also independent of the new name for the identifier. This means there is a simple way to compute the number of questions each tool will ask, given the name of the property to rename. For our approach, we can count the number of groups of related tokens with the given name, and for search-and-replace we can count the number of tokens. Tokens that refer to local variables are ignored in this statistic.

Renaming tools are typically invoked by selecting an identifier token in the editor and then choosing the rename action from a menu or using a shortcut. There is no reason to ask a question for this initially selected token, so we subtract one question per property name to represent this question that is answered for free. Thus, the tool may potentially ask zero questions to complete a refactoring. However, we shall disregard properties that are only mentioned once in the source code, so search-and-replace will by design always ask at least once. We also disregard the special property name `prototype` since it is easily recognized as a built-in property and is thus not renamed in practice.

To measure effort, we compare the number of questions issued per benchmark by search-and-replace versus our approach, simulating one rename refactoring on each distinct property name appearing in the benchmark. Application developers are unlikely to want to rename properties inside third-party libraries, and vice versa, so we separate the set of renaming tasks for application code and library code. However, we allow our refactoring tool to analyze the application and library code together as a whole, to avoid interference with the whole-program metric that we address in Section 4.4.

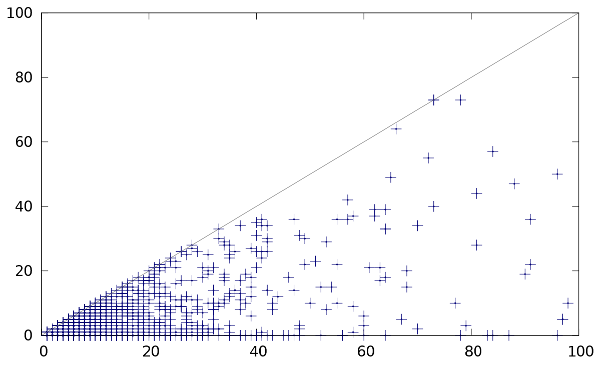

Figure 5. Number of questions asked by search-and-replace (x axis) versus our tool (y axis). Each point represents a property name in a benchmark.

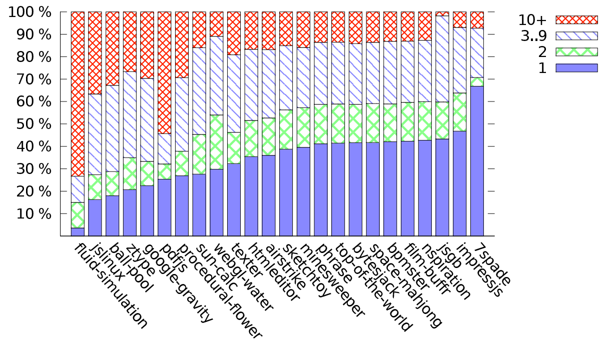

Figure 6. Percentage of identifier tokens (y axis) belonging to groups of size N (segment).

In Figure 5, each simulated renaming is plotted as a point positioned according to the number of questions asked by search-and-replace and by our tool, respectively. The diagonal highlights the place where the two tools are equally effective. The plot has been clamped to a limit of 100 questions, since the majority of points lie in this range. The most extreme outliers excluded from the plot are the property names `c` and `f` in *pdfjs*, which require 6,092 questions with search-and-replace and 14 questions with our tool. A total of 4,943 renamings were simulated in this experiment. Figure 6 shows how tokens are distributed in groups of various sizes during the simulated renamings.

To estimate the effect per benchmark, we sum the total number of questions for each property name in each benchmark. In Table 1, the *effect* column denotes the reduced programmer effort, based on this estimate. It is computed using this formula:

$$effect = \frac{\#search\text{-}replace\text{-}questions - \#our\text{-}questions}{\#search\text{-}replace\text{-}questions}$$

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

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

[7] `http://www.github.com/`

[8] `http://www.brics.dk/jsrefactor/renaming-benchmarks.zip`

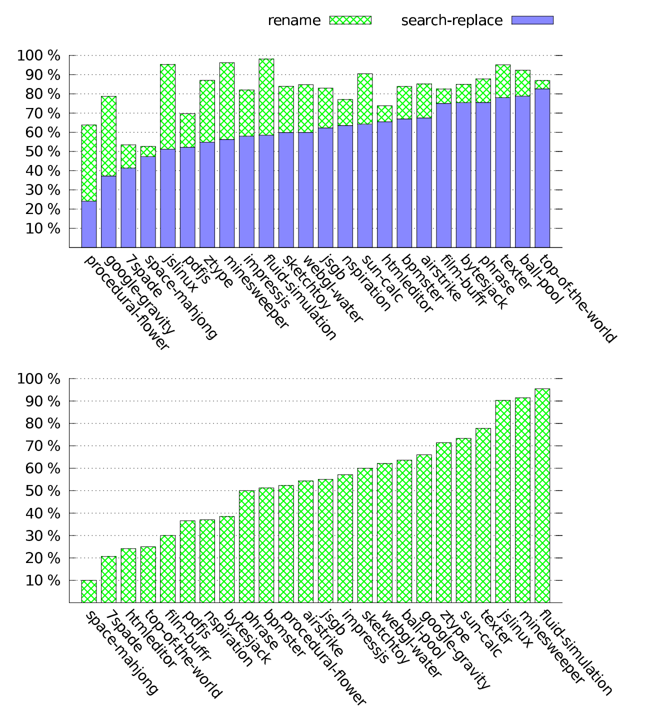

Figure 7. Number of property names (y-axis) renamable with ≤ 3 questions, relative to the number of candidate property names in each benchmark. The bottom graph focuses on the set of property names that search-and-replace cannot rename with ≤ 3 questions, showing how many can be handled with our technique using ≤ 3 questions.

Thus, if our tool asks the same number of questions as search-and-replace, we get an effect of 0%, while if we ask no questions at all, we get an effect of 100%.

The effect is in most cases between 40% and 70% but reaches 17% and 97% in extreme cases. The average and median effect are both approximately 57.1%. Two thirds of the benchmarks show an effect of more than 50%.

Each library that is used by multiple applications is analyzed together with each client application. In each case, we observe that the effect is exactly the same regardless of which application is used.

Although this indicates a significant reduction in the effort required to perform renamings, it may be that the number of questions remains so high that programmers deem the refactoring too costly and choose to omit an otherwise desirable renaming altogether. If we regard renaming tasks that require more than three questions as *hard* and renaming tasks that require at most three as *easy*, we can consider how many renaming tasks are easy when using search-and-replace versus our tool.

Figure 7 (top) shows how many properties can be renamed using at most three questions. All benchmarks benefit from the new technique, and in many cases we observe a substantial increase in the number of easy renamings. In Figure 7 (bottom) we focus on the property names for which search-and-replace requires more than three questions. This plot demonstrates that many of the renamings that are hard when using search-and-replace are easy when using our tool. For two thirds of the benchmarks, more than half of the hard renamings become easy.

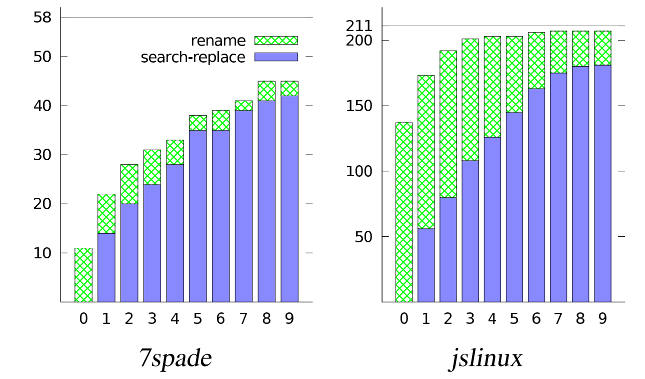

Figure 8. Number of property names (y-axis) renamable with $\leq N$ questions (x-axis). The horizontal line indicates the number of candidate property names in the benchmark.

In Figure 8 we vary the number of questions allowed, while focusing on two benchmarks, one with low and one with high effect. While the bound of three questions in Figure 7 was arbitrary, these figures indicate similar improvements for other bounds as well.

These figures indicate that even programmers who are unwilling to invest much time in renaming tasks may benefit from our tool. In summary, this part of the evaluation indicates a substantial reduction of programmer effort compared to traditional search-and-replace.

4.2 Soundness

As described in Section 3, our program analysis is theoretically unsound, but designed to work soundly on common programming patterns. We define a *failure* as a situation where unsoundness occurs in practice, that is, where our technique classifies two identifier tokens as related, even though they were not intended to refer to the same property. To estimate how likely this is to happen, we used a combination of dynamic analysis and manual inspection.

We exercise instrumented versions of each application, including the libraries they use, and record aliasing information at runtime. In these executions, we manually provide user input to each application, aiming to use all of its features at least once. High coverage is desirable, as it reduces the amount of manual inspection we must do afterward.

Given two expressions $e_1 . f$ and $e_2 . f$, if e_1 and e_2 evaluate to the same object at some point in the execution, then we have strong evidence to support that the two f tokens should indeed be related. If our technique classifies these as related, we therefore consider the classification as correct.

			whole		*isolated*		*fragmented*	
Applications	*lines*	*libraries used*	*effect*	*failure*	Δ*effect*	*failure*	Δ*effect*	*failure*
7spade	959		29.3%	-	-	-	-	-
airstrike	1,508	*prototype*	68.5%	-	-	-	−0.03 pp	-
bpmster	990	*jquery*	49.6%	-	−0.41 pp	-	−0.63 pp	X
bytesjack	685	*jquery*, *typekit*	52.7%	-	−0.15 pp	-	-	-
film-buffr	203	*jquery*, *typekit*	29.3%	-	-	-	-	-
nspiration	575	*jquery*, *typekit*	35.1%	-	−0.70 pp	-	−1.60 pp	-
phrase	176	*jquery*, *typekit*	60.4%	-	-	-	-	-
space-mahjong	344	*jquery*, *typekit*	17.2%	-	-	-	-	-
sun-calc	589		67.1%	-	-	-	-	-
top-of-the-world	158	*jquery*, *typekit*	49.1%	-	-	-	-	-
ball-pool	347	*box2d*, *prototype*	55.5%	-	-	-	-	-
fluid-simulation	484		97.0%	-	-	-	-	-
google-gravity	6,621		71.1%	-	-	-	−0.05 pp	-
htmleditor	1,835	*codemirror*, *esprima*	43.9%	-	-	-	-	-
minesweeper	562	*jquery*	90.8%	-	-	-	-	-
procedural-flower	2,215	*dat.gui*	67.9%	-	-	-	-	-
sketchtoy	959	*jquery*, *show_ads*	61.0%	-	-	-	-	-
texter	246	*dat.gui*	64.3%	-	-	-	-	-
webgl-water	1,627		57.4%	-	-	-	-	-
ztype	3,581	*show_ads*	72.3%	-	-	-	−0.06 pp	-
impress.js demo	448	*twitter_widgets*	57.1%	-	-	-	-	-
jsgb	3,815	*show_ads*	44.9%	-	-	-	-	-
jslinux	10,054		80.4%	-	-	-	−0.27 pp	-
pdfjs	38,958		72.5%	-	-	-	−0.02 pp	-
Libraries								
jquery	6,098		48.4%	-	-	-	−0.03 pp	X
typekit	1,359		51.1%	-	-	-	−0.04 pp	-
prototype	4,954		48.7%	-	-	-	−0.15 pp	-
box2d	6,081		72.1%	-	-	-	−0.03 pp	-
esprima	3,074		63.9%	X	-	(X)	-	(X)
codemirror	5,008		52.0%	-	-	-	−0.03 pp	-
dat.gui	2,149		58.8%	-	-	-	-	-
show_ads	590		54.1%	-	-	-	-	-
twitter_widgets	2,965		41.7%	-	-	-	−0.22 pp	-

Table 1. Experimental results. The effect column denotes reduced programmer effort (higher is better). The Δ*effect* columns are in percentage points (pp), relative to the effect column (zero is better). An 'X' denotes a potential failure discussed in the text. The symbol '-' indicates zero.

In situations where our tool classifies two tokens as related, but the dynamic alias information does not provide evidence supporting this relationship, we resort to manual inspection. This may happen because of incompleteness in the concrete execution, or more commonly, because of metaprogramming patterns in which properties are copied between objects, which our dynamic analysis cannot detect.

This experiment uncovers a single potential failure, which takes place in the *esprima* library—a JavaScript parser written in JavaScript. This library represents abstract syntax trees with objects, and all such objects have a `type` property denoting the type of the node it represents. Different types of nodes occasionally have same-named properties; a property name, such as, `value` is quite common among these. However, an *esprima* developer might hypothetically want to rename the `value` property for some AST nodes, but not all of them. Such a renaming could be done consistently, but our technique will not permit such a refactoring, because differently typed AST nodes are occasionally returned from within the same function, hence regarded as same-typed. This potential failure is marked with an X in the leftmost *failure* column in Table 1. Except for this single case, the experiments confirm that our analysis is sound in practice when the complete program code is available.

4.3 Delay

For each application, we measure the time required to analyze the application code together with its library code. We analyze each benchmark eleven times, discard the timings from the initial warm-up run, and record the average time of the other ten runs.

The analysis takes less than one second in every case, with 780 milliseconds for *pdfjs* being the slowest. Altogether, our implementation handles an average of around 50,000 lines per second on a 3.00 GHz PC running Linux.

When the user initiates a renaming, he must first enter a new name for the renamed property. The parsing and analysis do not depend on this new name and can therefore be performed in the background while the user is typing. As long as the user takes more than one second to type in the new name, there should therefore be no observable delay in practice when using our renaming technique.

4.4 Whole-Program

For the experiments described in the preceding sections, our static analyzer has an entire application available. However, library developers will typically not have a range of client applications ready for when they want to rename something.

To measure how well our tool works for library code, we repeat the experiment from Section 4.1, except that each library is now analyzed in isolation. As result, we observe no difference in the *effect* for any library compared to the previous experiment, in which the application code was included. This is indicated in the *isolated/*Δ*effect* column of Table 1.

For completeness, we also did the converse experiment: we analyze the applications without their libraries, even though application developers will typically have a copy of their libraries available. For three applications, the effect diminishes slightly, but never by more than a single percentage point. In the most pronounced case, the effect drops from 35.1% to 34.6% in the *nspiration* application.

These experimental results indicate that our approach is effective without a whole-program assumption.

We also want to support refactoring of code that is under development, which we call *incomplete* code. Such code might have radically different characteristics than finished code.

To estimate how well our tool works on incomplete code, once again we repeat the experiment from Section 4.1, except now with random pieces of code removed from each benchmark. We say the source code has been *fragmented*. Concretely, we replace random function bodies with empty bodies, which simulates incomplete code while avoiding introduction of syntax errors. For each benchmark, we produce ten fragmented versions randomly, each being roughly half the size of the original version. For each fragmented version, we then compute groups of related identifier tokens and compare these with the corresponding groups from the original version. We then record which token pairs were no longer considered related after fragmentation and compute the difference in effect accordingly (while only considering tokens that were not removed).

As result, the effect diminishes for seven applications and six libraries, in each case only slightly. The largest change is for a fragmented version of *nspiration*, in which the effect drops from 69.8% to 66.7% (again, for identifier tokens that were not removed in the fragmentation). The *fragmented/*Δ*effect* column in Table 1 shows the reduction in effect, averaged over the ten fragmented versions. The numbers demonstrate that our analysis is robust, even when major parts of the code are omitted.

This experiment also exposes a type of unsoundness caused by our namespace detection mechanism discussed in Section 3.4. The benchmarks *bpmster* and *jquery* both contain constructor functions in namespace objects. After fragmentation, all the `new` calls that allowed us to classify these functions as constructors were occasionally deleted, causing the receiver type inference phase to treat the constructors as methods. This in turn causes some unrelated tokens to be classified as related. As discussed in Section 3.4, the namespace detection uses a whole-program assumption, which surfaces in these few cases where the relevant code is not available to the analysis. In principle, these failures could be averted by allowing the programmer to provide a single namespace annotation to each benchmark, marking a few global variables as namespaces.

4.5 Threats to Validity

The validity of these encouraging experimental results may be threatened by several factors. Most importantly, the simulated refactorings may not be representative of actual use cases. An alternative evaluation approach would be to conduct extensive user studies, but that would be major endeavour in itself, which is also the reason why most literature on refactoring algorithms settle for automated experiments.

The fragmented code we produce might not be representative of code under development. Revision histories from version control systems might provide a more faithful representation of incomplete code, but we did not have access to such data for most of our benchmarks.

Finally, our selection of benchmarks might not represent all mainstream JavaScript coding styles, although they have been selected from different sources and vary in complexity, functionality, and use of libraries. All our benchmarks are browser-based. However, our approach works without any model of the host environment API, so it seems reasonable that it should work well also for other platforms.

5. Related Work

In previous work [3], we showed that points-to information can be used to perform various refactorings on JavaScript programs, fully automatically and with safety checks to ensure that program behavior is preserved. That technique, however, has some practical limitations regarding scalability and whole-program assumptions of the points-to analysis, as discussed in the introduction. Our new more pragmatic approach requires more manual effort from the user, but applies to a wider range of codebases.

Fast alias analyses in the style of Steensgaard [11] use union-find data structures to efficiently find aliased expressions in a program. Our type inference system is inspired by this kind of analysis, but as discussed in Section 3, it is technically not an alias analysis, since we occasionally want non-aliased expressions to have the same type.

Several related static analysis techniques have been developed specifically for JavaScript. The type analysis by Jensen et al. [6] is based on flow-sensitive dataflow analysis, Vardoulakis [13] uses a pushdown flow analysis to infer type information, and the flow analysis by Guha et al. [5] relies on reasoning about control flow. The notion of types we use here is different, and the application of our analysis is not type checking but refactoring. Sridharan et al. [10] have made advancements toward analyzing library code with points-to analysis, and Madsen et al. [7] have devised a practical technique for analyzing application code without including library source code or modeling external APIs. None of all these approaches scale to JavaScript programs of the size we consider here. Moreover, these analyses are conservative in the direction opposite of what we want: what they classify as spurious flow translates into unsoundness in our setting, and vice versa.

6. Conclusion

We have presented a technique for semi-automatic refactoring of property names in JavaScript programs, based on a static analysis for finding related identifier tokens. The analysis is easy to implement, and our experiments demonstrate that it is fast enough to be usable in IDEs. By simulating renaming tasks, our technique reduces the manual effort required to rename object properties by an average of 57% compared to the search-and-replace technique in existing JavaScript IDEs. This substantial improvement diminishes only slightly when used on incomplete code, and seemingly not at all when used on libraries without client code.

Although the analysis is theoretically unsound, the experiments show that it is sound in practice, except for a few rare cases, most of which could be eliminated entirely with simple source code annotations.

Our technique is based on a set of typing rules reminiscent of alias analysis. While we have focused on renaming so far, we would like to explore further applications of these typing rules in future work, such as, other refactorings, code completion, and documentation, to provide additional IDE support for JavaScript programmers.

Acknowledgements This work was supported by the Danish Research Council for Technology and Production. We thank Max Schäfer for his comments on a draft of this paper.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

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

[3] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2011.

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

[5] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proc. 20th European Symposium on Programming (ESOP)*, 2011.

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

[7] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. MSR-TR-2012-66, Microsoft Research, 2012.

[8] E. R. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Trans. Software Eng.*, 38(1):5–18, 2012.

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

[10] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proc. 26th European Conference Object-Oriented Programming (ECOOP)*, 2012.

[11] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1996.

[12] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proc. 34th International Conference on Software Engineering (ICSE)*, 2012.

[13] D. Vardoulakis. *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. Ph.D. thesis, Northeastern University, 2012.