Refactoring Tools for Dynamic Languages

Max Schäfer
IBM T.J. Watson Research Center
mschaefer@us.ibm.com

Abstract

Dynamic languages play an increasingly prominent role in modern software development. They are used in domains as diverse as web programming and scientific computing, for developing simple scripts as well as large applications.

Tool-supported refactoring for these languages can bring important benefits to programmers. First, manual refactoring tends to be error-prone in dynamic languages since they impose very little static structure; hence, erroneous refactorings often cannot be detected until runtime. A tool can use static program analysis to check the soundness of a proposed refactoring, thus mitigating this problem. Second, dynamic languages tend to lack constructs for modularisation and encapsulation, which can be an obstacle to writing maintainable software. In many cases, such constructs can be emulated using other language features, but refactoring a program to make use of such patterns is often non-trivial and could benefit from tool support. Third, refactoring tools can be useful for adapting programs to use high-level features, thus supporting script to program evolution.

However, specifying and implementing refactorings for dynamic languages is a challenging endeavour. We highlight some of the major issues in this area, and discuss recent progress towards solving them.

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

Keywords Refactoring, refactoring tools, dynamic languages, static analysis

1. Motivation

In modern software development, dynamic languages are ubiquitous: Perl and Python have long enjoyed great popularity as general-purpose scripting languages; Ruby and PHP dominate server-side web programming, with a recent survey indicating that the latter is used by about 77% of all websites [11]; on the client side, JavaScript (with its dialect ActionScript) is the only language that is supported across all major browsers, and it is the driving force behind Web 2.0; for scientific computing, R and MATLAB are widely used—and the list could be continued.

One important trait shared by these languages is their focus on rapid software development with dynamic typing and runtime name resolution, obviating the need for type declarations or static name binding. Additionally, they mostly favour a permissive execution model in which runtime errors are avoided as far as possible. For instance, reading a non-existent property of an object in JavaScript does not result in an error but returns the special value `undefined`, and writing to such a property will silently create it.

These features make manual refactoring more difficult than in static languages. Consider, e.g., an erroneous application of the RENAME FIELD refactoring for Java shown in Fig. 1. In the original program on the left, class `Rectangle` has a field `w` which is to be renamed to `width`. Assume we manually perform this renaming, yielding the program on the right where we forgot to update the reference to `w` on line 22. The access `this.w` is now unbound, resulting in a compiler error that quickly directs us towards the source of the mistake.

The situation is different in a dynamic language like JavaScript. Figure 2 shows a JavaScript version of the program from Fig. 1, where we now want to rename property `w` referenced on line 30 to `width`. If we forget to update the reference on line 34, as in the program on the right, the refactored program is still accepted by the interpreter.

Even running it will not result in an error: the access `this.w` on line 45 now refers to a non-existent property, hence it evaluates to `undefined`. Multiplying `undefined` by `4` (the value of expression `this.height`) also has a perfectly well-defined semantics: it yields the special value `NaN`, which is then returned from function `test`. If the program has a comprehensive test suite, the changed return value of `test` will be discovered, but this is not directly indicative of the inconsistent refactoring. In a large program, the point of runtime failure could be arbitrarily far removed from the point where a mistake was made during refactoring.

WRT June 01 2012, Rapperswil, Switzerland.

```
1  class Rectangle {
2      private int w, height;
3      public Rectangle(int w, int h) {
4          this.w = w;
5          this.height = h;
6      }
7      public int area() {
8          return this.w * this.height;
9      }
10     public static int test() {
11         Rectangle r = new Rectangle(3, 4);
12         return r.area();
13     }
14 }
```

```
15 class Rectangle {
16     private int width, height;
17     public Rectangle(int w, int h) {
18         this.width = w;
19         this.height = h;
20     }
21     public int area() {
22         return this.w * this.height; // error: no field "w"
23     }
24     public static int test() {
25         Rectangle r = new Rectangle(3, 4);
26         return r.area();
27     }
28 }
```

Figure 1. An inconsistent RENAME FIELD refactoring in Java, detected by the compiler (changes highlighted in grey)

```
29 function Rectangle(w, h) {
30     this.w = w;
31     this.height = h;
32 }
33 Rectangle.prototype.area = function() {
34     return this.w * this.height;
35 };
36 function test() {
37     var r = new Rectangle(3, 4);
38     return r.area();
39 }
```

```
40 function Rectangle(w, h) {
41     this.width = w;
42     this.height = h;
43 }
44 Rectangle.prototype.area = function() {
45     return this.w * this.height;
46 };
47 function test() {
48     var r = new Rectangle(3, 4);
49     return r.area();
50 }
```

Figure 2. An inconsistent RENAME PROPERTY refactoring in JavaScript, detected at runtime (changes highlighted in grey)

This example suggests that manual refactoring is even more error-prone in dynamic languages than in static languages. A robust refactoring tool that automatically handles precondition checks and transformations could thus be highly beneficial for programmers.

Another notable feature of many dynamic languages is their lack of modularity constructs such as packages or namespaces, or access control rules.

While such constructs may not be necessary for writing small throw-away scripts, dynamic languages are increasingly used for developing larger applications: a recent study found that many popular websites use several megabytes of script code [7], and the Swedish pension system famously relies on a 350,000-line Perl program [12].

For large programs, the lack of modularity constructs can become a liability. Indirect evidence for this is provided by the popularity of books such as "JavaScript: The Good Parts" [1], which propose patterns for emulating access control, modules and similar features, and recommend their use for improving code maintainability and reusability.

An example is the module pattern shown in Fig. 3. Here, two functions `Circle` and `Rectangle` are declared inside a closure that provides a common lexical scope for them. An object literal containing references to both functions is then constructed on line 70 and stored into the global variable `geometry`. This variable acts like a module in that `Circle` and `Rectangle` must now be accessed as `geometry.Circle` and `geometry.Rectangle`, respectively, as shown on lines 76 and 77. Inside the module "body", on the other hand, no such

```
51 var geometry = (function() {
52   function Circle(r) {
53     this.radius = r;
54   }
55   Circle.prototype.area = function() {
56     return Math.PI * this.radius *
57            this.radius;
58   };
59   Circle.prototype.containingRect = function() {
60     return new Rectangle(2*this.radius,2*this.radius);
61   };
62
63   function Rectangle(w, h) {
64     this.width = w;
65     this.height = h;
66   }
67   Rectangle.prototype.area = function() {
68     return this.w * this.h;
69   };
70   return {
71     Circle: Circle,
72     Rectangle: Rectangle
73   };
74 })();
75
76 var r = new geometry.Rectangle(3, 4),
77     c = new geometry.Circle(1);
```

Figure 3. Emulating modules in JavaScript

qualification is necessary as shown on line 60. Any declaration that is not stored in the object literal assigned to the module variable will effectively become private to the module and is inaccessible from outside.

The process of extracting a number of top-level declarations and statements into a module closure of this form is very naturally thought of as a refactoring, EXTRACT MODULE [2]. While the transformation needed to achieve this

```
78 ndims = size(n);
79 mdims = size(m);
80 isCompatible = ((length(ndims) == 2) && ...
81                 (length(mdims) == 2) && ...
82                 (ndims(2) == mdims(1)));
```

Figure 4. MATLAB script that checks whether two matrices are compatible for multiplication

```
83 function r = MultiplyCompatible(n, m)
84   ndims = size(n);
85   mdims = size(m);
86   r = ((length(ndims) == 2) && ...
87        (length(mdims) == 2) && ...
88        (ndims(2) == mdims(1)));
89 end
```

Figure 5. MATLAB function version of script in Fig. 4

refactoring looks deceptively simple, there are in fact a whole range of subtle semantic issues to deal with, which we will not explain here in detail. One particularly thorny question is determining whether or not an access to a module member is safe: if it is lexically outside the module closure, the access needs to be qualified, but this is only safe if the module can be guaranteed to have been constructed by the time the access happens. Since JavaScript supports higher-order functions, this is not an easy property to guarantee.

A third area where refactoring tools would be beneficial to developers using dynamic languages is in eliminating reliance on deprecated language features or adapting programs to use high-level constructs.

For instance, programs in MATLAB often consist of a collection of scripts sharing a common global namespace. An example of such a script is shown in Fig. 4: when called from another script, it assumes that variables `n` and `m` are already defined, and then checks whether `n` and `m` are matrices that are compatible for multiplication (i.e., `n` has as many columns as `m` has rows); the result of this check is stored in `isCompatible`, overwriting any previous value that a caller may have stored in that variable.

MATLAB also provides functions as shown in Fig. 5. Functions declare their input and output parameters (`n`, `m` and `r` in this case), which are local to the function, as are any previously undeclared variables defined within the function body, such as `ndims` on line 84 and `mdims` on line 85.

While scripts are convenient for incrementally developing programs using MATLAB's read-eval-print loop, larger programs obviously benefit from using functions instead.

Converting a script into a function is again a refactoring. Apart from determining suitable input and output parameters, this refactoring needs to deal with some very subtle issues involving name resolution: in MATLAB, function calls and array element accesses use the same syntax; the disambiguation algorithm is quite involved, and it differs between functions and scripts. For programmers unfamiliar with the details of this algorithm it would be all too easy to accidentally change its outcome when converting a script to a function. A refactoring tool that implements the full disambiguation algorithm, on the other hand, can relatively easily guarantee the correctness of the transformation [6].

A similar class of refactorings addresses overuse of low-level language features. A recent survey of the use of the `eval` function in JavaScript [8] found that in many cases uses of `eval` could be replaced by other, more high-level language constructs such as higher-order functions or reflective property accesses, potentially improving clarity, performance and even security of the code. While it may be possible to perform such refactorings by hand, a tool that automatically detects opportunities for these refactorings and verifies their safety would still be a great help to programmers.

2. Challenges

The previous section has argued that refactoring tools for dynamic languages can be useful both for automating refactorings like RENAME that are similar to their counterparts in static languages, and for supporting novel refactorings specifically tailored to dynamic languages.

Yet, there are significant challenges that have to be addressed in order to create a robust and useful refactoring tool for a dynamic language. We will briefly outline some of the major challenges in specifying and implementing refactorings for JavaScript and similar languages.

When specifying the preconditions and transformations for JavaScript refactorings, the main challenge is the lack of static program structure. Take, for example, a well-known Java refactoring like RENAME FIELD or PULL UP METHOD. Intuitively, the corresponding JavaScript refactorings would be renaming a property of an object and pulling a method up into an object's prototype object. But what, precisely, should these refactorings do?

In Java, when renaming a field we usually want to rename all references to that field, but leave references to other fields of the same name alone. In JavaScript, this distinction is not as clear: properties are not declared, instead they are created on first write; at runtime, a property access `x.f` simply looks up the relevant property on the object that `x` evaluates to.

In particular, two property accesses `x.f` and `y.f` may at one point during the execution refer to the same property on the same object, while at other times referring to properties of different objects. It is thus not a priory clear what set of runtime objects should be affected by the renaming (in Java, it would be all objects inheriting the renamed field).

Similarly, while the class hierarchy of a Java program is static and unchanging, the prototype hierarchy of a JavaScript program is completely dynamic and may change at runtime. This makes it hard to come up with simple specifications of refactorings such as PULL UP METHOD.

One possibility explored in recent work on JavaScript refactorings is to use static program analysis as a surrogate for the missing static program structure [2]. For instance,

when renaming a property access `x.f`, any other access `y.f` will be renamed along with it if the analysis determines that `x` and `y` may potentially alias at runtime. An unfortunate side effect is that the precise meaning of a refactoring dependens on the precision of the underlying analysis.

On the implementation side, it turns out that for many dynamic languages the static information needed by a refactoring tool is difficult to obtain. For instance, if the abovementioned specification of renaming is implemented, the question of whether a given access must be renamed, which is a simple static lookup question in Java, becomes a pointer aliasing problem. Likewise, since many dynamic languages support higher-order functions, call graphs can in general only be constructed using points-to analysis.

Scalable points-to analysis is a difficult problem even for static languages, but it is even more difficult for dynamic languages, where many commonly used type-based filtering techniques for improving analysis precision are not applicable. Dynamic languages generally also provide a rich reflection API, usually including an `eval` function for dynamically compiling and executing source code at runtime. This makes sound pointer analysis all but impossible in the general case.

Other challenges for refactoring tools for dynamic languages are of a more sociological nature. For instance, the use of IDEs into which refactoring tools are usually integrated does not appear to be very widespread yet among programmers using dynamic languages, making it harder for a refactoring tool to attract users.

3. Work on Refactoring for Dynamic Languages

Refactoring tools for dynamic languages are almost as old as refactoring itself: the very first refactoring tool, the Refactoring Browser [9], targeted the dynamically typed object-oriented language Smalltalk. In contrast to JavaScript, Smalltalk does require declarations for fields and methods and has a static class hierarchy, which simplifies automated refactoring. The Refactoring Browser in fact does not perform static type analysis, but uses a dynamic analysis to determine the runtime types of expressions. It hence requires a comprehensive test suite to work well.

The JSRefactor project has investigated the specification and implementation of refactorings for JavaScript based on static analysis [2]. Currently, this approach still suffers from scalability issues: the pointer analysis can only handle small programs, and in particular is unable to analyse applications based on popular JavaScript frameworks such as jQuery.

Research on static analysis for JavaScript is a fairly active research area [3, 4], however, with recent work reporting some success in analysing framework-based programs [10], so it is possible that a static analysis-based approach will become viable in the future.

The *Unevalizer* tool by Jensen et al. [5] integrates with a sophisticated data flow analysis to automatically eliminate uses of `eval`. However, their goal is to improve analysis precision rather than program structure, so their tool does not, in general, make a program more readable or maintainable.

There has also been some work on a refactoring tool for MATLAB [6], mostly concentrating on cleanup refactorings towards high-level constructs. The analysis problems here seem less formidable than for JavaScript.

Initial refactoring support for JavaScript, Ruby and PHP has recently appeared in the Eclipse, IntelliJ and NetBeans IDEs, but so far only a few basic refactorings are supported and the implementations seem to mostly rely on heuristics rather than principled program analysis.

4. Conclusions

We have argued that software developers stand to benefit from refactoring tools for dynamic languages, both because they automate well-known refactorings that are more difficult to perform by hand than with static languages, and because they can support novel refactorings addressing issues specific to these languages.

Recent work both on program analysis and on refactorings for dynamic languages has shown promising first results, but significant challenges remain.

References

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

[2] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported Refactoring for JavaScript. In *OOPSLA*, pages 119–138, 2011.

[3] S. Guarnieri and V. B. Livshits. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, 2009.

[4] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *SAS*, 2009.

[5] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the Eval that Men Do. In *ISSTA*, 2012.

[6] S. Radpour and L. Hendren. Refactoring MATLAB. Technical Report No. 2011-2, McGill University, October 2011.

[7] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *PLDI*, pages 1–12, 2010.

[8] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval That Men Do. In *ECOOP*, pages 52–78, 2011.

[9] D. Roberts, J. Brant, and R. E. Johnson. A Refactoring Tool for Smalltalk. *TAPOS*, 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 *ECOOP*, 2012. To Appear.

[11] W^{3}Techs. Usage Statistics and Market Share of PHP for Websites. `http://w3techs.com`, 2012.

[12] T. Wrigstad, P. Eugster, J. Field, N. Nystrom, and J. Vitek. Software Hardening: A Research Agenda. In *STOP*, pages 58–70, 2009.