Evolution of Dynamic Feature Usage in PHP

Mark Hills
East Carolina University, Greenville, NC, USA
mhills@cs.ecu.edu

***Abstract*—PHP includes a number of dynamic features that, if used, make it challenging for both programmers and tools to reason about programs. In this paper we examine how usage of these features has changed over time, looking at usage trends for three categories of dynamic features across the release histories of two popular open-source PHP systems, WordPress and MediaWiki. Our initial results suggest that, while features such as `eval` are being removed over time, more constrained dynamic features such as variable properties are becoming more common. We believe the results of this analysis provide useful insights for researchers and tool developers into the evolving use of dynamic features in real PHP programs.**

I. Introduction

Starting as a collection of scripts for building personal homepages, PHP is now one of the most popular programming languages for building server-side web applications. Like other scripting languages, PHP includes a number of dynamic language features, such as an `eval` expression to run code provided at runtime as strings; special methods (referred to as *magic methods*) that handle accesses of object fields and uses of methods that are either not defined or not visible; and the ability to use expressions, instead of literal identifiers, to give the names of variables, functions, methods, classes in `new` expressions, and properties (referred to here as *variable features*, and specifically as, e.g., *variable functions* or *variable methods*). While these features often make the programmer's job easier, they make it harder for program analysis tools to statically compute a precise approximation of possible runtime behaviors. This, in turn, makes it harder to provide developers with powerful tools for working with large code bases, such as tools for finding possible security bugs and for refactoring.

In earlier work [1] we looked at how often dynamic features are used across a corpus of 19 popular open-source PHP systems. However, this work did not explore trends over time in the use of these features. These trends can be important for making reasonable assumptions about the code to be analyzed and for deciding where work on providing more precise approximations of runtime behavior will have the greatest impact. In this paper, we start by focusing on two of the systems from our original corpus, WordPress and MediaWiki, exploring how uses of variable features, magic methods, and `eval` have changed over time. Our initial results show that the use of some variable features, such as variable properties, is increasing, meaning these features will show up more often in analyzed code, while the use of others is decreasing or holding steady; magic methods are now more common, but are still (in an absolute sense) uncommon; and uses of `eval` are fortunately rare and becoming rarer.

The rest of the paper is organized as follows. In Section II we describe the corpus used in this paper, while Section III provides details on how the study was conducted. Section IV then describes our results, illustrating trends in feature usage and exploring the causes behind some of the changes. We then briefly discuss related work in Section V, focusing on empirical studies of software systems used to improve program analysis tools, while Section VI discusses future work and concludes. All software used in this paper is available for download at https://github.com/ecu-sle-lab/saner-2015.

TABLE I. The Corpus, First and Last Releases.

System	Version	PHP	Released	File Count	SLOC
WordPress	0.71	4.0.6	June 9, 2003	54	11,613
	4.0	5.2.4	September 4, 2014	481	138,414
MediaWiki	1.1.0	4.3.2	December 8, 2003	146	45,230
	1.23.5	5.3.2	October 1, 2014	1,629	294,731

II. Corpus

Although we plan to analyze release histories for all systems used in our prior study [1], we opted to start with two: WordPress and MediaWiki. Both systems are popular, with extensive release histories spanning a number of years and PHP versions. The corpus used in this paper consists of the entire release histories of both systems as of October 1, 2014, excluding non-stable releases such as alpha, beta, and release candidate-quality releases. One exception is the first WordPress release; since version 0.70 is no longer available, the first version used is version 0.71, listed as a "gold" release. Table I shows information on the first and last releases of each system in the corpus, including the version number, required PHP version,[1] date of release, file count, and lines of code. In total, 93 releases of WordPress, consisting of 6,966,378 lines of code and 25,732 files, and 189 releases of MediaWiki, consisting of 83,045,245 lines of code and 167,588 files, are included.

III. Research Method

All analysis performed in this paper was performed using Rascal [2], a meta-programming language for source code analysis and transformation which uses a familiar Java-like syntax and is based on immutable data (trees, relations, sets), term rewriting, and relational calculus primitives. Scripting the analysis ensures that the results are reproducible and checkable. Rascal was also used to generate the figures in this paper.

To get the code for each release of the studied systems, the current versions of the GitHub repositories for both WordPress [2] and MediaWiki [3] were cloned. The tags in each repository were

[1] 4.3.2 is not required for the first version of MediaWiki, but the distribution mentions that earlier versions have known problems.

[2] https://github.com/WordPress/WordPress, last checked on January 23, 2015.

[3] https://github.com/wikimedia/mediawiki, last checked on January 23, 2015.

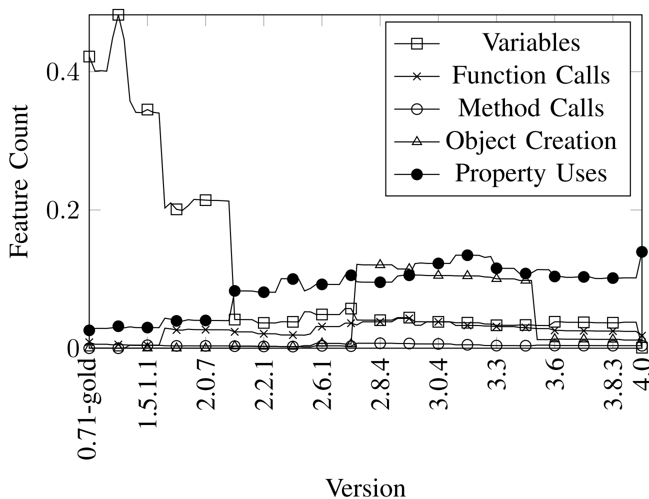

Fig. 1. Variable Features in WordPress, Scaled by SLOC.

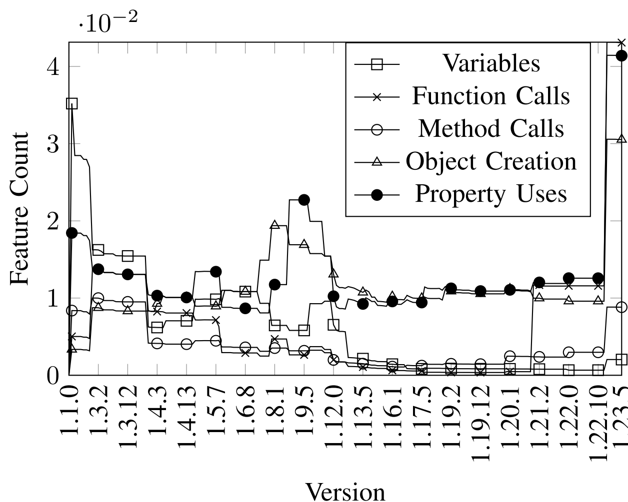

Fig. 2. Variable Features in MediaWiki, Scaled by SLOC.

then manually filtered to keep only normal releases, and a script was written to checkout and copy the source for each release into a separate directory, named for the version. The code in WordPress was analyzed unchanged, while the only change to MediaWiki was to rename occurrences of `Namespace` in earlier releases, since it now clashes with a reserved word in PHP. All files in each release were then parsed using our fork [4] of an open-source PHP parser.[5] This parser generates ASTs as terms formed over Rascal's algebraic datatypes, which are then serialized for later analysis, with Rascal locations linking abstract constructs back to the related concrete constructs.

The analysis code, written in Rascal, uses these serialized ASTs to build summaries for each release. These summaries include all the counts used in the analysis described in this paper except for the lines of code and files counts; these are extracted separately using cloc,[6] stored in a CSV file, and read in as relations using Rascal's resources functionality [3]. Feature counts are calculated by Rascal using pattern matching over ASTs. Distribution graphs are then directly generated based on the computed counts, which are normalized by lines of code. The latter ensures that trends seen in the graphs are not just natural increases in feature counts caused by an increase in the size of the system over time.

```
1  // 1. /includes/deferred/SiteStatsUpdate, line 62
2  $update->$field = $deltas[$field];
3
4  // 2. /includes/Skin.php, line 1644
5  function __call( $fname, $args ) {
6    $realFunction = array( 'Linker', $fname );
7    if ( is_callable( $realFunction ) ) {
8      return call_user_func_array( $realFunction, $args );
9    }
10 }
11
12 // 3. /maintenance/eval.php, line 31
13 $val = eval( $line . ";" );
```

Fig. 3. Examples of Dynamic Features in MediaWiki 1.23.5.

IV. Evolution of Dynamic Feature Usage

Our analysis focused on three classes of dynamic features in PHP: variable features, magic methods, and eval-like constructs.

A. Variable Features

PHP includes 10 variable constructs which allow an arbitrary expression to be used in place of an identifier. Here, we focus on five of these constructs: variable variables, where the variable name itself is given using an expression; variable function and method calls, where the function or method name to call is given as an expression; variable object creation/new, where an expression is used to give the name of the class in a `new` expression; and variable properties, where an expression is used to give the name for a property used in a property set or fetch. An example of the latter is shown in Figure 3 on line 2. Here, an array named `$deltas` maps names to values; this array is used to set properties of the same name (`$field`) on the object `$update`. The surrounding loop, not shown, iterates over all the fields, setting each on the object in turn. The other five variable features are not included here since they occur so rarely: taken together, the high-water mark for these features is 1 in WordPress and 6 in MediaWiki, with current versions having 1 and 5, respectively, all related to static method calls.[7]

Figures 1 and 2 show the trend in usage of these five features for WordPress and MediaWiki, respectively. The x axis in these, and all other, figures gives the version numbers for the first and last releases, as well as every fifth (for WordPress) or tenth (for MediaWiki) release after the first in the release history. The y axis in these, and all other, figures then gives the count of the number of occurrences of a specific feature, scaled by source lines of code (SLOC). The legend indicates which feature corresponds with which plot in the graph.

The main trend to notice here is the increasing importance of variable properties. Other results are mixed: variable variable uses are lower and decreasing over time in both systems, but are still the most common variable feature in other systems

[4] https://github.com/cwi-swat/PHP-Parser
[5] https://github.com/nikic/PHP-Parser/, last checked on January 23, 2015.
[6] http://cloc.sourceforge.net, last checked on January 23, 2015.
[7] This is based on data from the extracted summaries, available at https://github.com/ecu-sle-lab/saner-2015.

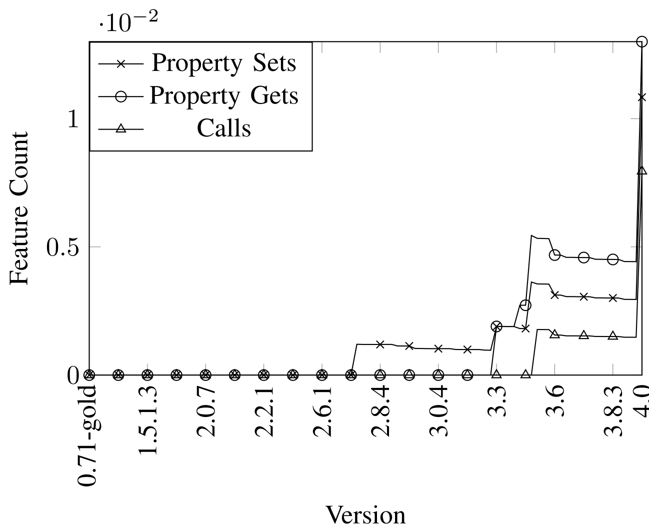

Fig. 4. Magic Methods in WordPress, Scaled by SLOC.

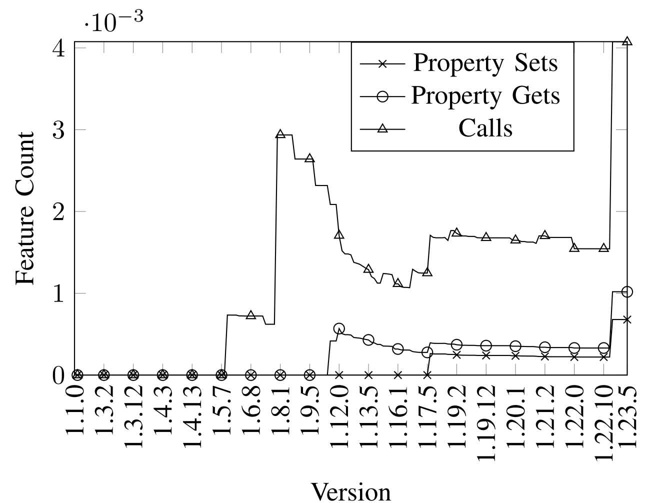

Fig. 5. Magic Methods in MediaWiki, Scaled by SLOC.

(e.g., phpMyAdmin), so we know this isn't universal. Other variable features also show significant differences between the two systems, with variable functions and object creations being relatively more popular in MediaWiki than in WordPress. This indicates that, although support for all five of these variable features needs to be included to analyze realistic programs, improving support for handling variable properties may become more critical over time and when analyzing newer code.

Although we have yet to examine the code to determine the causes of all the sudden changes shown in the graphs, we have looked specifically at two: the sudden decrease in variable object creations in WordPress between versions 3.3 and 3.6, shown in Figure 1, and the sudden increase in variable function calls in MediaWiki around version 1.22.10, shown in Figure 2. For the first, this is caused by a major refactoring of SimplePie, an RSS and Atom feed framework used by WordPress. This accounts for almost the entire change in object creations in the graph. For the second, the sudden increase in variable function calls is caused mainly by the use of variable functions inside PHPUnit tests added to MediaWiki. Without this there is still an increase, but it is much more modest. We believe this last one is especially significant: assuming we are mainly interested in the code actually run as part of the site, it will be important in further studies to separate test code and regular application code when possible so as not to possibly bias the results.

B. Magic Methods

In PHP, magic methods are defined to provide generic handlers for undefined or non-visible properties and methods. PHP includes six kinds of magic methods, which handle property gets and sets; method calls; static method calls; and calls to either `isset` or `unset` on properties. An example of a magic method for handling calls in shown in Figure 3 on lines 5 through 10. This method takes the name of the called method as `$fname` and checks to see if a static method of this name is callable on class `Linker`. If so, this static method is called with the arguments in `$args`.[8] Our analysis focuses on the first three kinds of magic methods; the last three do not occur in MediaWiki at all and are rare in WordPress, although recent increases for `isset` (5 in 3.9.2, 17 in 4.0) and `unset` (2 in 3.9.2, 14 in 4.0) warrant further study.

Figures 4 and 5 show the trend in usage for these three features for WordPress and MediaWiki, respectively. In MediaWiki, magic methods for handling method calls are the most common, with property gets and sets relatively rarer, but trends for both are relatively stable over most of the last third of the release history. The spike at the end is mainly caused by a decrease in the SLOC for newer versions of MediaWiki, making all magic methods relatively more common. For WordPress, there instead has been an increase in use of magic methods since between versions 3.3 and 3.6, with a marked recent increase at version 4.0 (15 sets, 18 gets, and 14 calls, plus the support for `isset` and `unset` mentioned above). However, the absolute numbers are still quite small. Further analysis, identifying usage sites for these features in the code, is required to determine the actual impact of these additions; this analysis will require type inference support for PHP, currently being developed as part of our work on PHP AiR. In systems, such as WordPress, where magic method usage becomes more common, type analysis will become more important for the precision of other analyses, such as call graph construction.

C. Eval-Like Constructs

PHP includes an `eval` expression that, at runtime, is given a string containing PHP code which is then evaluated. Similar functionality is provided by `create_function`, which accepts two arguments, a string representing a list of arguments and a string representing the function body, and uses these to create an anonymous (i.e., nameless) function that can be used as a callback function or invoked as a variable function. Figure 6 and 7 show how usage of these features has evolved for WordPress and MediaWiki, respectively, while an example of an eval is shown in Figure 3 on line 13. This eval, part of the maintenance console, runs an arbitrary PHP command given as input to the console.

These figures show a clear decrease in usage of both `eval` and `create_function` in WordPress after an earlier

[8]Some additional code, to issue a deprecation warning and, if the static method does not exist, to throw an exception, is not shown.

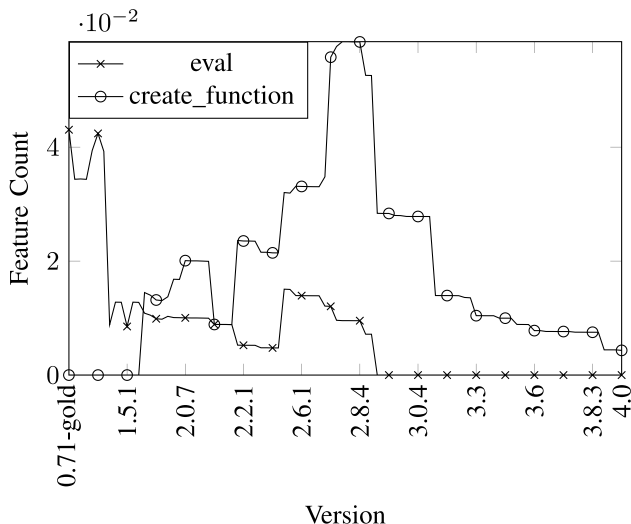

Fig. 6. Eval Constructs in WordPress, Scaled by SLOC.

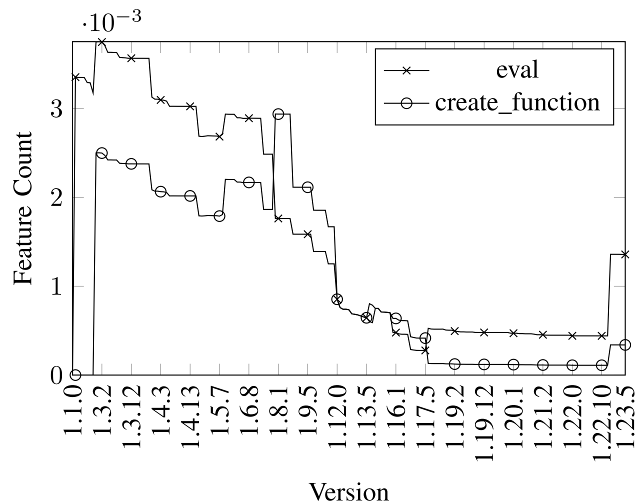

Fig. 7. Eval Constructs in MediaWiki, Scaled by SLOC.

increase. In MediaWiki usage is holding steady in a range that has never grown above five for either feature, and has totaled five for both across roughly a third of the version history, with no uses in the main user-facing code (`create_function` is used in maintenance code to dump wiki information, while `eval` is used in unit and parser testing code and in maintenance code to allow console access to site information). The recent decrease in SLOC for MediaWiki does make these features relatively more common in the newest releases, however, accounting for the increase at the end. For WordPress, there were 8 evals in version 2.8.4, which then went down to 6 in the next release, and to 0 in version 2.9.0. Comparing versions 2.8.4 and 2.9.0, 6 of these were used for generating function calls in file `class-pclzip.php`, which supports processing ZIP archives, and were replaced with callbacks,[9] while the other 2 were replaced with standard function calls. There were also 49 uses of `create_function` in version 2.8.4, which went down steadily to the current number, 6, as of version 4.0. Comparing these two versions, the main change is that uses of `create_function`, used to provide callback functions for `preg_replace_callback` (for regular-expression based search and replace, 9 occurrences), `array_filter` (for filtering array elements, 5 occurrences), `array_map` (for mapping a function over an array, 2 occurrences), and various other functions for sorting, filtering, and adding notices (5, 5, and 2 occurrences, respectively), were replaced with callbacks. In 2 cases, explicit callbacks were removed, and in the rest this code has either been moved or is no longer in WordPress.

Based on these numbers, and results from other systems we have looked at [1], these features seem to be rarely used, becoming less popular as other language features replace them. These newer features are easier to analyze—the callbacks noted above are almost always given using literal function or method names—so effort spent on trying to precisely analyze constructs like `eval` (e.g., using string analysis techniques) could generally be spent more profitably elsewhere.

[9]A *callback* in PHP is the name of a function, an array containing an object instance and a method name, or an array containing a class name and a method name for static calls. In newer versions of PHP, closures can also be used.

D. Threats to Validity

The main threat to validity is that, so far, we have only fully analyzed the release histories for two systems, WordPress and MediaWiki. Other systems may display trends that are quite different. However, we don't believe that this will be the case. The results presented here line up well with the results of our earlier work [1], where (for instance) we showed that `eval` and `create_function` are not commonly used in PHP systems. The most likely differences, therefore, will be with changes in trends related to variable features. Even there, though, showing that variable properties are becoming more common would still be true, even if not across all systems, and results for the other variable features already appear to be quite system-specific. As additional systems are analyzed, similar graphs, as well as all extracted counts, will be posted on the project website, linked at https://github.com/ecu-sle-lab/saner-2015.

V. Related Work

There are a number of studies focused on using static techniques to examine how language features are used or to find patterns that can be exploited in analysis. As part of their work on alias analysis, Hackett and Aiken studied aliasing patterns in large C systems programs [4], identifying nine patterns that account for almost all aliasing encountered in their corpus. Ernst et al. investigated usage of the C preprocessor [5], with these results then used in further experiments related to preprocessor-aware C code analysis and transformation, such as that by Garrido [6]. Liebig et al. instead conducted a targeted empirical study to uncover the use of the preprocessor in encoding variations and variability [7]. Collberg et al. performed an in-depth empirical analysis of Java bytecode [8], computing object-oriented (e.g., classes per package, fields per class) and instruction-level (e.g., instruction frequencies, common sequences of instructions) metrics. Baxter et al. focused on characterizing the distributions for a number of metrics computed over Java bytecode and Java source code [9].

Other studies have used a combination of static and dynamic analysis, or have focused just on using dynamic analysis. Knuth used static and dynamic techniques to examine real-world

FORTRAN programs [10], gathering statistics over FORTRAN source code and using profiling and sampling to gather runtime information. Richards et al. used trace analysis to examine how the dynamic features of JavaScript are used in practice [11], investigating whether the scenarios assumed by static analysis tools (e.g., limited use of `eval`, limited deletion of fields, uses of functions that match the provided function signatures) are accurate. In a more focused study over a larger corpus, Richards et al. then analyzed runtime traces to find uses of `eval` [12], categorizing these uses into a number of patterns. Morandat et al. undertook an in-depth study of how the features in the R language are actually used [13], using runtime tracing and static analysis to examine a corpus of 3.9 million lines of R code. Furr et al. used profiling to determine how the dynamic features of a Ruby program are used in practice [14]. They discovered that these features are generally used in ways that are almost static, allowing them to replace many uses of dynamic features with static variants that are amenable to static type inference in a system such as DRuby [15].

The work in this paper is built directly on our earlier work on feature usage in PHP systems [1], which looked broadly at feature usage in single releases of 19 different open-source systems, and makes use of PHP AiR [16], a framework for PHP program analysis in Rascal [2]. A study by Kyriakakis and Chatzigeorgiou [17] also looked at the evolution of PHP systems, including WordPress. However, instead of looking at how dynamic features are used, they focused on aspects of the systems directly impacting maintainability, such as the use of libraries, occurrences of unused code, the adoption of newer object-oriented language features, and the stability over time of the interfaces provided by the system.

VI. Conclusions and Future Work

PHP provides a number of dynamic language features, making it challenging for both programmers and tools to reason about programs making use of these features. In this paper, we have highlighted the changing trends in the use of several of these features in two systems, WordPress and MediaWiki. In these systems, uses of `eval` and `create_function` are decreasing over time, both in an absolute and—except for the spike in MediaWiki caused by the decrease in SLOC—a relative sense; magic methods are either becoming increasingly common, or holding at their current level of use; and the use of variable properties is increasing, something that needs to be taken account of in program analyses that reason about the fields of objects.

Although we believe that our initial results already provide some useful feedback for both developers and researchers, there are further opportunities to expand this research. One is to expand the number of systems studied. We have already begun doing so, starting with the release history of phpMyAdmin, and plan to continue with the rest of the systems in our original corpus. Another is to expand the range of features being explored, including features such as dynamic invocation, which provides support for invoking functions and methods by name—similar to the callbacks used to replace calls to `create_function`, discussed in Section IV. The results of these additional investigations should provide further information to developers and researchers, both about what features to use or (potentially) avoid and what features need to be supported to provide precise program analysis algorithms supporting future releases of software systems.

Our longer term goal is to take advantage of these results in building effective program analysis tools for PHP. We have already started to do so based on our earlier results, creating an analysis for resolving dynamic file inclusion expressions that is effective in many cases [18]. Similarly, we plan to investigate patterns of use for variable properties that can be leveraged to increase the precision of the type analysis we are currently building for PHP, and we are investigating dynamic techniques for better dealing with variable features and dynamic function invocations, especially those related to plugins.

References

[1] M. Hills, P. Klint, and J. J. Vinju, "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective," in *Proceedings of ISSTA 2013*. ACM, 2013, pp. 325–335.

[2] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM 2009*. IEEE, 2009, pp. 168–177.

[3] M. Hills, P. Klint, and J. J. Vinju, "Meta-language Support for Type-Safe Access to External Resources," in *Proceedings of SLE'12*, ser. LNCS, vol. 7745. Springer, 2012, pp. 372–391.

[4] B. Hackett and A. Aiken, "How is Aliasing Used in Systems Software?" in *Proceedings of FSE 2006*. ACM, 2006, pp. 69–80.

[5] M. D. Ernst, G. J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[6] A. Garrido, "Program Refactoring in the Presence of Preprocessor Directives," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[7] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," in *Proceedings of ICSE 2010*. ACM, 2010, pp. 105–114.

[8] C. S. Collberg, G. Myles, and M. Stepp, "An empirical study of Java bytecode programs," *Software: Practice and Experience*, vol. 37, no. 6, pp. 581–641, 2007.

[9] G. Baxter, M. R. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. D. Tempero, "Understanding the Shape of Java Software," in *Proceedings of OOPSLA 2006*. ACM, 2006, pp. 397–412.

[10] D. E. Knuth, "An Empirical Study of FORTRAN Programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.

[11] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *Proceedings of PLDI 2010*. ACM, 2010, pp. 1–12.

[12] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications," in *Proceedings of ECOOP 2011*, ser. LNCS, vol. 6813. Springer, 2011, pp. 52–78.

[13] F. Morandat, B. Hill, L. Osvald, and J. Vitek, "Evaluating the Design of the R Language - Objects and Functions for Data Analysis," in *Proceedings of ECOOP 2012*, ser. LNCS, vol. 7313. Springer, 2012, pp. 104–131.

[14] M. Furr, J. hoon (David) An, and J. S. Foster, "Profile-Guided Static Typing for Dynamic Scripting Languages," in *Proceedings of OOPSLA 2009*. ACM, 2009, pp. 283–300.

[15] M. Furr, J. An, J. S. Foster, and M. W. Hicks, "Static Type Inference for Ruby," in *Proceedings of SAC 2009*. ACM, 2009, pp. 1859–1866.

[16] M. Hills and P. Klint, "PHP AiR: Analyzing PHP Systems with Rascal," in *Proceedings of CSMR-WCRE 2014*. IEEE, 2014, pp. 454–457.

[17] P. Kyriakakis and A. Chatzigeorgiou, "Maintenance Patterns of Large-Scale PHP Web Applications," in *Proceedings of ICSME 2014*. IEEE, 2014, pp. 381–390.

[18] M. Hills, P. Klint, and J. J. Vinju, "Static, Lightweight Includes Resolution for PHP," in *Proceedings of ASE 2014*. ACM, 2014, pp. 503–514.