Enabling PHP Software Engineering Research in Rascal

Mark Hills\textsuperscript{a}, Paul Klint\textsuperscript{b,c}, Jurgen J. Vinju\textsuperscript{b,c}

\textsuperscript{a}East Carolina University, Greenville, NC, USA
\textsuperscript{b}Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
\textsuperscript{c}INRIA Lille Nord Europe, Lille, France

Abstract

Today, PHP is one of the most popular programming languages, and is commonly used in the open source community and in industry to build large application frameworks and web applications. In this paper, we discuss our ongoing work on PHP AiR, a framework for PHP Analysis in Rascal. PHP AiR is focused especially on program analysis and empirical software engineering, and is being used actively and effectively in work on evaluating PHP feature usage, program analysis for refactoring and security validation, and source code metrics. We describe the requirements and design decisions for PHP AiR, summarize current research using PHP AiR, discuss lessons learned, and briefly sketch future work.

Keywords: meta-programming, program analysis, empirical software engineering, dynamic languages, PHP

1. Introduction

PHP\textsuperscript{[1]} invented by Rasmus Lerdorf in 1994, is an imperative, object-oriented language focused on server-side application development. It is now one of the most popular languages, as of February 2014 ranking 6th on the TIOBE programming community index\textsuperscript{[2]} used by 81.8 percent of all websites whose server-side language can be determined\textsuperscript{[3]}, and ranking as the 4th most popular language on GitHub by repositories created in 2013\textsuperscript{[4]}. This popularity has led to the creation of a number of large, widely-used open source applications and application...
frameworks, including WordPress, Joomla, Drupal, MediaWiki, Symfony, Magento and CodeIgniter, and has made it a popular choice for developers creating new web applications and frameworks.

The availability of such large, open-source systems provides an ideal ecosystem for empirical software engineering research. PHP is also a fascinating subject for program analysis research. Most PHP applications are web-based, giving an urgency to program analysis targeted at detecting potential security errors. At the same time, the dynamic nature of the language (e.g., duck typing, reflection, evaluation of code built at runtime using strings), as well as its use on larger and larger systems, increases the importance of analyses targeted at program understanding, automated code refactoring, and programmer tool support, all areas where PHP currently lags behind languages such as Java.

To enable research in program analysis, automated refactoring, tool support, and empirical software engineering in PHP, we are developing PHP AiR, an environment for PHP Analysis in Rascal. Built using the Rascal meta-programming language, a successor to Asf+Sdf and RSCRIPT, PHP AiR has been, and is currently being, used in multiple research projects: an empirical survey of PHP language feature usage, a program analysis for resolving dynamic file includes, a taint analysis for detecting dangerous uses of unchecked user-provided strings in PHP library calls, a refactoring from hand-coded HTML to uses of template libraries, a similar refactoring from hand-coded SQL calls to uses of database libraries, and multiple projects to extract various metrics from PHP source code.

The rest of this paper is organized as follows. In Section 2, we provide a brief introduction to Rascal. Section 3 constitutes the core of our paper, discussing the requirements and design decisions for PHP AiR, giving a high-level introduction to the tool, and presenting some of the research performed so far, which helps to motivate these decisions and indicates how they have worked in practice. Section 4 presents related work for Rascal and for the analysis of PHP programs, while Section 5 concludes, discussing lessons learned and future directions. The PHP AiR system is available online at https://github.com/cwi-swat/php-analysis while more information about Rascal is available at http://www.rascal-mpl.org/. A demonstration of PHP AiR, including a screencast, is available on SHARE at http://is.ieis.tue.nl/staff/pvgorp/share/.

http://wordpress.org/
http://www.joomla.org/
http://drupal.org/
http://www.mediawiki.org/wiki/MediaWiki
http://symfony.com/
http://www.magentocommerce.com/
http://ellislab.com/codeigniter
2. Rascal

Rascal was designed to cover the entire domain of meta-programming. The language itself is designed with unofficial “language layers.” This allows inexperienced Rascal programmers to start with just the core language features, adding more advanced features as they become more comfortable with the language. This language core contains basic data-types for booleans, integers, reals, source locations, date-time, lists, sets, tuples, maps, relations (sets of tuples), and list relations (lists of tuples); structured control flow, e.g., if, while, switch, for; and exception handling with try and catch. The syntax of these constructs is designed to be familiar to programmers: for instance, if statements and try/catch blocks look like those found in C and Java, respectively. All data in Rascal is immutable (i.e., no references are ever created or taken), and all code is statically typed. The built-in data type of source locations is particularly suited for creating references to source fragments as they appear during analysis and transformation of source code.

Rascal’s type system is organized as a lattice, with bottom (void) and top (value) elements. The Rascal node type is the parent of all user-defined datatypes, including the types of concrete syntax elements (e.g., Stmt and Expr). Numeric types also have a parent type, num, but are not themselves in a subtype relation (e.g., real is not a parent of int).

Beyond the type system and the language core, Rascal also includes a number of more advanced features. These features can be progressively used by the programmer to create more complex programs, and are needed in Rascal to enable the full range of meta-programming capabilities. These more advanced features include parameterized algebraic data type definitions; a built-in grammar formalism, which includes disambiguation facilities and annotatable grammar rules and is used to generate scannerless generalized top-down parsers; pattern matching over all Rascal types, including set matching, list matching, and deep matching (i.e., matching at an arbitrary depth within a term) over nested structures, as well as pattern-based dispatch for invoking Rascal functions; comprehensions over lists, sets, and maps; visit statements for performing structure-shy traversals (allowing the visit to match just those cases of interest, with default traversal behavior for the rest) and transformation of Rascal terms; powerful string templating capabilities; and a built-in notion of fixpoint computation. Rascal resources [6] provide access to external sources of data from within Rascal, leveraging the Rascal type system to ensure that uses of external data are well-typed and to provide more convenient access (e.g., by providing field names based on column names in a database table).

A number of Rascal features focus on the safety and modularity of Rascal code. While types of local variables can be inferred, parameter and return types in functions must be provided. This allows better error messages to be generated, since errors detected by the inferencer can be localized within a function, and also provides documentation (through type annotations) on function signatures. Also, the only casting mechanism is pattern matching, which prevents the problems with casts found in C (lack of safety) and Java (runtime casting...
exceptions). Finally, the use of persistent data structures eliminates a number of standard problems with using references which can leak out of the current scope or be captured by other variables.

3. PHP AiR: PHP Analysis in Rascal

PHP AiR is being built with certain high-level requirements in mind, and a number of design decisions have been made during development of the tool. Below we discuss these requirements and design decisions; provide a high-level overview of the tool, including illustrative examples of using PHP AiR for PHP code exploration; and discuss ongoing research using PHP AiR in the domains of empirical software engineering and program analysis.

3.1. Requirements

When building PHP AiR we had several core requirements. First, and most importantly, it should be possible to use PHP AiR to effectively and efficiently support empirical software engineering and program analysis research on real PHP systems. This means that scalability is important: as counted by clocl\(^{12}\), WordPress version 3.8.1\(^{13}\) is made up of 482 PHP script files with 132,877 lines of PHP code (excluding comments and whitespace), while MediaWiki version 1.22.2\(^{14}\) is 1,865 PHP script files with 1,036,960 lines of PHP code. Second, PHP AiR should be interactive to enable what-if analyses and exploratory programming. For instance, it should be possible to write queries over PHP code to find all uses of a given language feature, with support provided for jumping directly to these uses in the underlying source code. It should also be possible to prototype new program analysis tools and new visualizations supporting understanding of the code. Third, PHP AiR should support integration with a standard PHP development environment, allowing it to be used, in a familiar way, by the largest possible audience. Finally, given that Rascal has been developed in our group for exactly these kinds of applications, we wanted PHP AiR to profit as much as possible from Rascal’s language features and libraries, without excluding the use of external, non-Rascal tools (e.g., Eclipse-based PHP tools). Using Rascal’s extension mechanisms, these external tools can be made accessible by creating Rascal bindings to existing Java libraries, enabling direct access for interacting with these tools whenever that would be advantageous.

From our perspective as the developers of Rascal, these requirements provide an additional benefit not specific to PHP AiR: the development of PHP AiR provides valuable ongoing feedback for our development of Rascal, as well as a strong incentive to address any issues raised. Some of these issues are discussed below and in Section 5.

\(^{12}\)http://cloc.sourceforge.net
\(^{13}\)http://wordpress.org/news/2014/01/wordpress-3-8-1/
\(^{14}\)http://www.mediawiki.org/wiki/MediaWiki_1.22
3.2. Design Decisions

Although the creation of PHP AiR has involved a number of design decisions, we believe there are three that stand out, each of which is explored further below. First, what is the best way to parse the PHP source that we want to analyze within PHP AiR? Second, how heavily should Rascal be used in PHP AiR? And third, how can we take advantage of external sources of data within PHP AiR?

3.2.1. Parsing PHP

One of the earliest decisions we had to make when designing PHP AiR was how to parse PHP. One obvious solution would be to use Rascal: Rascal includes a grammar notation with built-in support for precedence, associativity, and filtering of unwanted parse trees; a parser generator which uses this notation to generate scannerless, generalized parsers; and direct support for pattern matching over concrete syntax terms. Rascal also includes a partial importer for SDF grammars, which would allow importing significant portions of an existing grammar included as part of the PHP-Front project.

While this option is still open, we decided not to use it initially. Instead, we parse PHP scripts using our fork of an open-source PHP parser. This parser, in turn, is based on the C parser included in the Zend Engine, the engine used to execute PHP programs. Our fork extends the original parser with added support for properly visiting PHP ASTs, tracking source locations, resolving magic constants, and generating Rascal term representations of PHP ASTs. Not only did using this parser provide a quicker start, the fact that it is based on the official grammar makes it easier to take advantage of new PHP language constructs as they are added to the language and provides additional assurance that our ASTs faithfully represent the underlying code.

One downside of this approach is that it does not provide a direct way to interact with PHP code inside an IDE, which is one of our requirements. To provide IDE support, we also built an integration layer between PHP AiR and the Eclipse PHP Developer Tools (PDT). This is similar to the integration that Rascal already has with the Eclipse Java Developer Tools (JDT), which we have been able to exploit in areas such as refactoring. Both approaches to parsing PHP target the same abstract syntax, allowing either to be used by the other tools developed in PHP AiR. To handle some inconsistencies in the internal representation used by the PDT (including some apparent errors), a normalization process is used to ensure the ASTs generated by the JDT-based...
parser are the same as those generated by the external parser, transforming them to match if necessary. While we have not verified that this is correct in all cases, we have tested this on thousands of PHP files, using our earlier work on PHP feature usage [4] to ensure we have covered all features of the language in current use. We also plan to work with the developers of the PDT parser to resolve these inconsistencies.

3.2.2. Rascal Usage in PHP AiR

PHP AiR, structured as a collection of Rascal modules, has been developed completely in Rascal except for the parsers, where our decision to allow multiple parsers has led to development in a combination of PHP, Java, and Rascal (discussed above). This has allowed PHP AiR to take advantage of many of the Rascal features mentioned in Section 2, especially features for data representation (e.g., algebraic data types, maps, relations) and data querying (e.g., pattern matching). PHP AiR currently consists of 13,037 lines of Rascal code and 4,668 lines of PHP code (1,507 to convert the AST to Rascal, as described above, the rest generated to provide a reasonable Visitor interface for the AST). In addition, the integration layer with the PDT consists of 83 lines of Rascal and 1,952 lines of Java (all counted using cloc).

This use of the built-in Rascal data types has led to some challenges, especially in our work on program analysis, and has led to one of our main design decisions. Most analysis algorithms work by extracting and iterating over a number of extracted program facts. Given the sizes of the programs we are working with, and the use within Rascal of persistent (immutable) data structures, this can lead to a large amount of memory churn. This led us to ask the question: should we use optimized data structures written in Java and made available through a Rascal interface, or use the built-in Rascal data types instead? The second provides cleaner, more flexible code, allowing the full range of Rascal operations to work over the data structures, while the first could provide improved performance when analyzing large systems by minimizing memory usage and the churn caused by continuous allocation, copying, and collection.

We initially started with the second approach, moved to the first, and have now returned to the second. Originally the built-in data types proved to use too much memory, motivating us to switch to using optimized data types written in Java and available through Rascal libraries. However, what we found is that the algorithms we use in the analysis are much more important to performance than the data structures used to store analysis information as the computation proceeds, which has led us to switch back to using the built-in Rascal types while also looking at faster algorithms and improvements to the datatype implementations, a process that is ongoing. Separate work on a compiler for Rascal also promises to further increase performance of PHP AiR.

3.2.3. Access to External Sources of Data

A third decision was to provide support, within Rascal, for accessing the types of external data needed for PHP AiR. This decision led to our ongoing research on Rascal resources [6], mentioned above in Section 2. Currently in
PHP AiR we mainly use resources to access data generated by external tools and stored in CSV files; importing these files as resources generates a Rascal relation for each file, with field names and types based on the column names and the type of the data stored in the column. The code needed to load the file contents from disk into the relation is also generated. Rascal currently supports read-only resources, but we are working on extending this work to support writing to resources as well as using more intelligent methods of reading from resources that will minimize memory use. In the future, in cases where optimized storage is needed for extracted facts, resources should allow us to store these facts in databases while, at the same time, providing a native Rascal interface to read and write these facts.

3.3. Installing and Using PHP AiR

PHP AiR is distributed through GitHub[^22] as an Eclipse project. Full installation instructions are given on the PHP AiR site. PHP AiR requires a parser—currently, either the external parser, written in PHP, or an Eclipse-based parser using the PDT—and a current version of Eclipse. Since Rascal Eclipse projects are also Eclipse plugins, PHP AiR can then be extended either by modifying the PHP AiR project directly or by creating a new Rascal project and including PHP AiR as a plugin dependency. It is also possible to use the existing functionality of PHP AiR without writing new Rascal code by using the existing functionality from the Rascal console.

Figure 1 gives a high-level overview of how PHP AiR is used. First, individual PHP files, or whole systems (e.g., WordPress), are parsed and converted into

[^22]: https://github.com/cwi-swat/php-analysis
Rascal terms representing ASTs. These ASTs are then the base structure used by all other operations in PHP AiR. Figure 2 shows several examples of this process. parsePHPExpression invokes the external PHP parser, returning an AST representing the expression, given as a string. In the first example, the expression 1+2 is parsed, generating an AST node representing binary operations. This node has three children: the two scalars, 1 and 2, and plus(), indicating the operation used. In the second example, a simple function call, f(3,4), is parsed, generating a call node with three children: the function name (which, in PHP, could also be an expression) and the two parameters. false in each parameter indicates that neither is a reference parameter. The third example shows the result of loading an entire system (with the full location elided to ensure it fits on the page). This produces a result of type System, which is a map from locations (of the underlying files) to ASTs representing each script.

Operations in PHP AiR are divided into two general categories (which, in reality, can sometimes overlap): interactive code querying and empirical analysis on one hand, program analysis on the other. Both are explored further below.

3.4. Empirical Software Engineering

When using PHP AiR for interactive code querying and empirical analysis, the user, using Rascal, can write queries over the PHP system, supplemented by external sources of data, and can also script these queries, aggregate results, and use various standard Rascal functions for statistical analysis. Results are computed and then written either to the console or to an external format, such as a table or figure in a \LaTeX document or a .dot file representing a graph. An example of a query is given in Figure 3 (with source locations shortened so they fit on the page): a deep match identifies all function calls in WordPress 3.8.1, with a regular expression match used to constrain this to only those calls to functions with mysql in the function name. The result is a relation from the location of the call to the actual call expression, a portion of which (here, a call to function mysql2date) is shown. Although better handled by machines
rascal> cls = { < c@at, c > | /c:call(name(name(fn)),_) := wp381, /mysql/ := fn }; rel[Loc,Expr]: { 
  <.../wp-includes/class-wp-editor.php(31963,45,824,0,824,0),call( 
    name(name("mysql2date")[ 
      @at=.../wp-includes/class-wp-editor.php(31963,10,824,0,824,0) 
    ], 
    [ actualParameter( call{
      ... 
  ) 

Figure 3: Searching for calls to SQL related functions.

than by humans, one can see source locations at work here: they consist of a standard URI between vertical bars ("|") and may be followed by precise line and column information to describe a text range in the referenced URI.

Our current research has focused on using empirical software engineering techniques [9] to examine feature usage in large open-source PHP systems [4]. To start, we assembled a corpus of 19 large open-source PHP systems, basing our choice on popularity rankings provided by Ohloh [23], a site that tracks open-source projects. In total, the corpus consists of 19,816 PHP source files with 3,370,219 lines of PHP source (counted using the cloc [24] tool). An extension to this corpus, looking specifically at dynamic file includes, adds 20 additional systems selected from the GitHub PHP page [25], specifically from the most starred and most forked repositories for the day, week, month, and overall, with the goal of ensuring that more “regular” code (i.e., not just large, well-maintained projects) was also covered. This added an additional 15,492 files with 1,805,333 lines of PHP source.

Using this corpus, we focused on both general characteristics of PHP programs, such as the size of PHP files and the distribution of PHP language features, as well as on the use of dynamic features in the code. For the latter, we looked at how often these features occur in practice, how distributed these features are in PHP programs, and also how often these dynamic features are actually static in practice, meaning that static techniques can be used to reason about these features and minimize their impact on program analysis tools. Interesting findings include that eval is rarely used in practice (only 148 times in total in the original corpus), that variable variables (variables that contain the name of another variable, allowing indirect access) can, in many cases, be resolved statically to a specific set of referenced variables, and that many new features such as goto and traits are not yet used in popular systems (no uses at all were found in the corpus for either feature). At the same time, ongoing student projects and industry collaborations are using similar techniques to derive metrics from PHP code.

This work has led to performance improvements in Rascal to better handle

---

23 http://www.ohloh.net/tags/php
24 http://cloc.sourceforge.net
large quantities of PHP system data, and has helped us improve the built-in statistics libraries. It has also helped us improve PHP AiR to support easier querying over language constructs as well as simpler ways of aggregating results to present online summaries and generate \LaTeX{} tables and figures.

3.5. Program Analysis

When using PHP AiR for program analysis, users can run either the predefined analysis functions that come with PHP AiR or create their own. Analysis results can be used to transform existing programs in the system, to supplement interactive queries or empirical analysis, or to display results. We are working on extending this display capability to take advantage of Eclipse, for instance by allowing warnings or errors computed by an analysis to be flagged directly in the PHP source files.

Our ongoing work on program analysis using PHP AiR is focused on a number of analysis tasks common to other programming languages as well as several that are more specific to PHP. For the first, we are building type inference and alias analysis passes that will provide information useful for programmer tools and other analysis passes. For the second, we are working on a taint analysis to detect possible security violations in calls to system functions (e.g., uses of unchecked user input to construct database queries), and are also working on the analyses needed to handle various refactoring operations, including a string analysis that will be needed to convert uses of HTML and SQL strings, built using string-building operations, into safer uses of HTML and SQL libraries.

We are also actively improving the precision of an analysis to determine which files are actually included by dynamic file inclusion expressions \[\text{[5]}\]. An example of the current “quick resolve” method of resolving includes, which performs a lightweight analysis based on constants and string matching and is intended to be used in cases where performance is important (such as in IDEs), is shown in Figure \[\text{[4]}\]. The first two lines (including a comment) show the include
to be resolved, which uses a constant imported from another file and string concatenation to build the name of the file to include. The next two Rascal statements define two variables, baseLoc and upload, which hold the location for the root of WordPress 3.8.1 and for the file containing the includes that need to be resolved. The call to quickResolve then performs the resolution, which uses an extracted model of the system, including information on all constants defined in the system (including those with unique definitions, meaning they are defined by the same literal along all possible include paths), to indicate the file or files that could be included by a given include expression. An example of the result, given as a relation from the location of the include expression to locations of possibly included files, is then shown—here, we can determine that there is a unique intended file included by this expression.

Although some of the challenges we faced are based on the dynamic nature of PHP, other challenges are caused by the size of the systems we are analyzing, along with the fact that Rascal’s memory consumption is not yet optimized. Practically, this means that we need to focus quite heavily on efficient, modular algorithms, and are also continuing to reduce the memory footprint and increase overall performance of Rascal. On the other hand, features of Rascal such as source locations have proven to be quite valuable, providing a way to easily tie back error information derived from ASTs to specific points in the source code. The availability of data types for maps and relations has also provided natural ways to represent many of the facts needed during program analysis.

4. Related Work

The design of Rascal is based on inspiration from many earlier languages and systems. The syntax features (grammar definition and parsing) were initially directly based on SDF [2], but the notation has changed, the expressivity has increased (c.f., earlier work discusses this evolution [10]) and we are at the moment developing a new parsing technology. The features related to analysis are mostly based on relational calculus, relational algebra and logic programming systems such as Crocopat [11], Grok [12] and RSCRIPT [3], with some influence from CodeSurfer [13]. Rascal has strongly simplified backtracking and fixed point computation features reminiscent of constraint programming and logic programming systems like Moreau’s Choice Point Library [14], Prolog and Datalog [15]. Rascal’s program transformation and manipulation features are directly inspired by term rewriting/functional languages such as ASF+SDF [16], Stratego [17], TOM [18], and TXL [19]. The ATerm library [20] inspired Rascal’s immutable values, while the ANTLR tool-set [21], Eclipse IMP [22] and TOM [18] have been an inspiration because of their integration with mainstream programming environments.

A number of related tools have been developed for the analysis of PHP
programs. The PHP-sat\textsuperscript{26} and PHP-tools\textsuperscript{27} projects include limited support (mainly intraprocedural) for security analysis as well as analyses to detect a variety of common bug patterns (e.g., assigning the result of a function that does not contain a return statement). PHP AiR is targeting more complex PHP programs and a wider variety of analyses. More focused tools include PHP CodeSniffer\textsuperscript{28} which checks PHP code for violations of defined coding standard, and the PHP Copy/Paste Detector\textsuperscript{29} which provides a very limited form of clone detection (i.e., only type 1 clones, which are exact textual copies, potentially with differences in whitespace). There are also several tools for calculating metrics for PHP code, including PHPDepend\textsuperscript{30} and PHPLoc.\textsuperscript{31} We are incorporating similar functionality, with hopefully better performance—PHPLoc is fast, but gives limited information, while PHPDepend is more complete but is quite slow when run on larger codebases. PHPMD\textsuperscript{32} both computes metrics and tries to find a number of programming flaws and potential bugs, but focuses mainly on areas that do not require sophisticated analysis.

Biggar \textsuperscript{23} with phc and Zhao et al.\textsuperscript{24} with HipHop both perform analysis (under various assumptions about the code) as part of the task of compiling PHP code, while Huang et al.'s WebSSARI\textsuperscript{25} and Jovanovic et al.'s Pixy system\textsuperscript{26, 27} use a combination of static analysis and (in the case of WebSSARI) program instrumentation to protect against security vulnerabilities. In Minamide’s work\textsuperscript{28}, the output of a PHP script is represented as a context-free grammar which over-approximates the resulting HTML document. This grammar is then used to check for well-formedness of the generated HTML and to look for cross-site scripting attacks, which can be detected by seeing if strings with problematic constructs are part of the language of the grammar. A similar approach is taken by Wassermann and Su\textsuperscript{29}, who use context-free grammars to represent sets of string values, finite-state transducers to represent string operations, and regular expressions to represent the security policy being enforced. Samirni et al.\textsuperscript{30} use a combined static/dynamic approach to check the HTML generated by a PHP program, ensuring it is not malformed. Instead of using context-free grammars, they use a combination of testing and string constraints.

While we are providing an environment with PHP AiR where similar analyses can be created, we are also looking at a number of novel analyses, including one to detect bugs introduced by changes to the semantics of PHP which occur as the language continues to evolve.

\textsuperscript{25}http://www.program-transformation.org/PHP/PhpSat\textsuperscript{26} http://www.program-transformation.org/PHP/PhpTools\textsuperscript{27} http://pear.php.net/package/PHP_CodeSniffer\textsuperscript{28} https://github.com/sebastianbergmann/phpcpd\textsuperscript{30} http://pdepend.org/\textsuperscript{29} https://github.com/sebastianbergmann/phploc\textsuperscript{30} http://phpmd.org/
5. Lessons Learned and Future Directions

From our experience building PHP AiR we have been able to extract a number of important lessons:

- Rascal’s high-level data types (e.g., sets, maps, relations, and ADTs) and language features (pattern matching, traversal, local backtracking, comprehensions) all favor a declarative programming style where the distance between a published algorithm and its Rascal implementation is often surprisingly small. This enables easy experimentation at the algorithmic level. It also leads to a significantly (up to an order of magnitude) smaller code size for the resulting tools compared to similar tools implemented in more traditional programming languages.

- Location information is important for research looking at language features and critical for program analysis, where it can be used to provide accurate messages. As already mentioned earlier, Rascal includes a source location data type as a language feature, with locations being used extensively in the libraries for IDE integration (e.g., for error reporting and source code annotation). This simplifies the processing of references to source code fragments.

- Flexibility in secondary tool choices, such as parsing, has been key to quickly getting PHP AiR off the ground. Since all code in PHP AiR works over AST nodes supplemented with location information, different parsers can be used as front-ends without needing broader changes to the source code of PHP AiR itself.

- The ability to script empirical analyses, and even to script the generation of artifacts for research papers such as tables and figures, has been an important feature, allowing us to make the results of our research reproducible in a form that can be more easily checked.

- The availability of Rascal resources has provided a clean way to reuse data created by external tools, and should be extended to encompass additional data sources.

- Performance is an ongoing concern, especially when analyzing larger and larger systems, and needs to be addressed in PHP AiR and directly within Rascal. There are still a number of cases where memory use or execution performance are unsatisfactory, but, more positively, these cases are driving improvements in the algorithms used in PHP AiR and in the implementation of Rascal.

We have a number of future plans for further developing and utilizing PHP AiR. First, we plan to complete integration with the Eclipse PHP Development Tools, allowing analysis and transformation of the code to be directed from within Eclipse.
Second, we plan to continue making improvements to the performance of Rascal to allow us to handle larger codebases. This includes compilation of Rascal to the JVM and optimization of the implementation of the built-in datatypes.

Third, we are working on $M^3$, a Metrics Meta-Model \cite{M3} that has as goal to model and store the results of program analysis. $M^3$ is inspired by models such as FAMIX, RSF, GXL, ATerms and S-Expressions. The differences are that $M^3$ deals with purely immutable, typed data and can be directly produced, manipulated and analyzed using Rascal primitives. Two unique elements are the introduction of source location literals to identify source code artifacts in a language agnostic manner and support for fully structured type symbols. $M^3$ itself is language-agnostic but can be extended to model language-specific features. A mature Java extension exists and extensions for other languages (including PHP) are under development. We plan to utilize persistent storage to more easily create, store and use extracted $M^3$ models.

Fourth, we also plan to continue work on program analysis and empirical software engineering using PHP AiR, which is proving to also be a good environment for student projects.

In the longer term, we plan to raise the level of abstraction in creating the various components of PHP AiR even further: we are working on domain-specific languages for tasks such as intermediate language generation and control flow graph construction. These DSLs should be usable for creating tools for languages other than PHP as well.

References


URL http://arxiv.org/abs/1312.1188