Understanding Database Usage in PHP Systems: Current and Future Work

Mark Hills (@hillsma on Twitter, http://www.cs.ecu.edu/hillsma/)

The Southeast Regional Programming Languages Seminar (SERPL 2019)
May 11, 2019
Augusta, GA, US

http://www.rascal-mpl.org

Why look at PHP applications?

- Popular with programmers: #7 on TIOBE Programming Community Index, behind Java, C, Python, C++, Visual Basic, and C#, and #4 by number of GitHub repos created as of 2018

- Used by large number of websites, including the roughly 1/3 of the web that run WordPress

- Lots of large systems, very few high-quality analysis tools

- Hostile environments: most PHP code runs on the web

What are we trying to do?

- Big picture: develop a framework for PHP analysis
- Specifics:
  - Empirical software engineering
  - Software metrics
  - Program analysis (static/dynamic)
  - Developer tool support

Rascal: The Meta-Programming One Stop Shop

- “Rascal is a domain specific language for source code analysis and manipulation a.k.a. meta-programming.” (http://www.rascal-mpl.org/)

- Language focus: program analysis, program transformation, domain-specific language creation

- Current projects across large numbers of domains, both within and outside academia

- Open source, committers worldwide

Rascal Benefits (Not an Exhaustive List!)

- Built-in language support for matching & transforming code
- Rich data types: relations, maps, lists, sets, tuples, parse trees, higher-order functions
- Console supports interactive exploration
- Extensible with Java and Eclipse
- Empirical research support: code querying, statistical analysis, interaction with external data (e.g., code repositories, external databases), visualization

PHP AiR Design Points

- Support for multiple PHP parsers

- Want to support integration with Eclipse (part way there), potentially other IDEs like PHPStorm, maybe use LSP

- Perform tasks by writing Rascal code (not focused on push-button solutions, goes with Rascal “no magic” principle)

- Want to work with real PHP code (WordPress, MediaWiki, etc)

PHP & SQL: Motivation

- We want to transform uses of older database APIs into equivalent uses of newer, safer APIs

- We want to aid developers in understanding how queries are built in their (maybe unfamiliar to them!) programs

- (For us and other researchers/tool builders) We want to better understand how PHP, database APIs, and query languages are used in existing code to help us build better tools for program analysis, comprehension, and transformation

Code Context

- MySQL API uses query functions (mysql_query) to execute queries
- Queries given as strings, formed using string building operations
- Queries often have a mixture of static and dynamic pieces

```
$query = mysql_query("
SELECT title
FROM semesters
WHERE semesterid = $_POST[semester]
");
```

Note: Real but horrible query, this has a major security vulnerability…

Research questions

- R1: How can we model how queries are built?

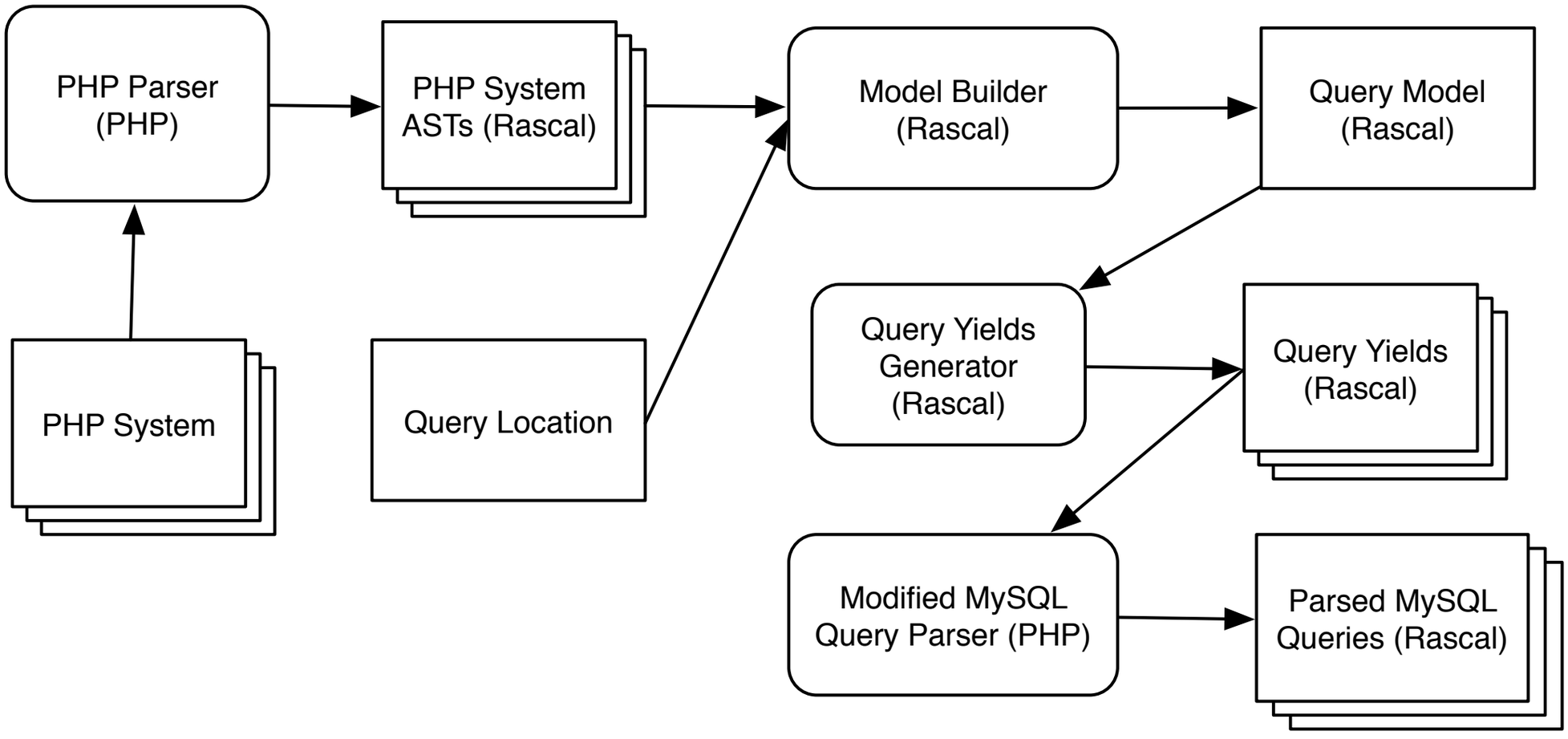
- R2: Using these models, how can we generate the queries or query templates (with placeholders for dynamic bits) that could actually be executed?

- R3: What parts of the query language (here, SQL) are used in these queries? Which of these parts are static, and which are dynamic?

Analyzing and parsing queries: methodology

- All analysis code is written using Rascal (http://www.rascal-mpl.org/), a meta-programming language for program analysis
  - https://github.com/ecu-pase-lab/mysql-query-construction-analysis
- PHP is parsed using a PHP parser written in PHP, parser yields Rascal terms
- MySQL queries are parsed using a fork of the parser found in phpMyAdmin, a web frontend for administering MySQL
- A quick note: this is all part of the PHP AiR project

Analyzing and parsing queries: “The Big Picture”



R1: Building models

Input : *sys*, a PHP system, mapping from file locations to abstract syntax trees
Input : *callLoc*, a location indicating the query call to be analyzed
Output : *res*, a query model

1 *inputCFG* ← `buildCFG4Loc` (*callLoc*)
2 *inputNode* ← the CFG node from *inputCFG* representing the call at *callLoc*
3 *d* ← `definitions` (*inputCFG*)
4 *u* ← `uses` (*inputCFG*, *d*)
5 *slicedCFG* ← `basicSlice` (*inputCFG*, *inputNode*, `usedNames` (*u*, *inputNode*), *d*, *u*)
6 *startingFragment* ← `expr2qf` (*inputNode*)
7 *fragmentRel* ← {}
8 **while** *fragmentRel* `continues to change` **do**
9 | *nodesToExpand* ← (*inputNode.l* × *startingFragment*) ∪ *fragmentRel*⟨3, 4⟩
10 | **foreach** (*nodeLabel* × *nodeFragment*) ∈ *nodesToExpand* **do**
11 | | add `expandFragment` (*nodeLabel*, *nodeFragment*, *d*, *u*) to *fragmentRel*
12 | **end**
13 **end**
14 *fragmentRel* ← `addEdgeInfo` (*fragmentRel*, *slicedCFG*)
15 *res* ← the model, including *fragmentRel*, *startingFragment*, and *callLoc*

Algorithm 1: Extracting a Model of a SQL Query.

See the paper for all the details, here comes a summary…

R1: Building models

- Models are (possibly cyclic) graphs of *query fragments* (with a bit of bookkeeping info, like the location of the call)

- A query fragment is a static or dynamic piece of the query

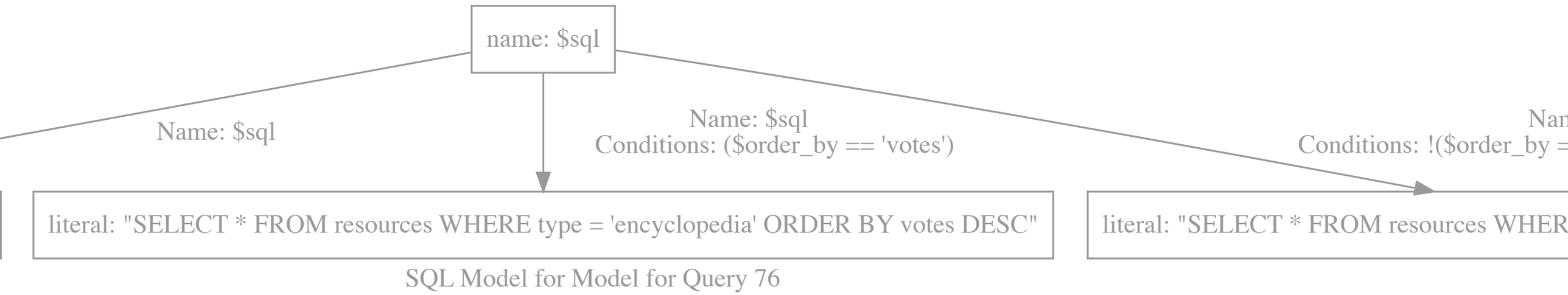
- Intraprocedural, backwards slice throws away code that does not impact query

- CFG and def/use info link names in fragments to defs of those names

- Reachability conditions used to decorate graph edges

Demo 1…


BUILD


name: $sql
Name: $sql
Name: $sql
Conditions: ($order_by == 'votes')
Nam
Conditions: !($order_by =
literal: "SELECT * FROM resources WHERE type = 'encyclopedia' ORDER BY votes DESC"
literal: "SELECT * FROM resources WHER
SQL Model for Model for Query 76

R2: Extract yields

- Yields are lists of “pieces”:
  - Static pieces for query text
  - Dynamic pieces for arbitrary expressions
    - Name pieces for names (useful to track separately)
- Generated by traversing the graph, currently cuts off when cycles detected
- Can use edge labels to filter infeasible yields, improving to work in more situations (loops are problematic)

Demo 2…

R3: Parsing partial queries

- First, yields are converted to strings: static pieces yield strings directly, dynamic and name pieces are turned into query holes (e.g., ?1, ?2, generally ?n)

- Second, string is parsed by our modified MySQL parser, yielding PHP objects representing MySQL AST (limitation: we assume holes are expressions and do not cross clause boundaries, supporting this is ongoing work)

- Third, AST pretty-printed to a Rascal term representing AST, similarly to current PHP parser

Demo…

And now for some controversy

- We want to extend this to be interprocedural,
  but: for a really dynamic language, where even the decision of what code to include is deferred until runtime, is this even useful?

- To borrow from earlier: keep it simple! Do we even need to support the entire language for this to be useful for developers?

- For artifacts, are full VMs at all useful? Should we aim at using something like Docker? Images available in the cloud? Something else?

Future Directions: Expand the Corpus!

- Earlier corpus was somewhat ad-hoc, based on past work, mix of current and no longer maintained systems

- Current corpus based on 1000 most starred repos on GitHub as of April 2018 (based on GHTorrent and BigQuery), identified 78 of these systems that use MySQL or MySQLi libraries

Future Directions: More APIs!

- Current work has focused on MySQL API
- Already (mostly) expanded to include MySQLi API
  - Challenge — type inference…
- Planning to add support for PDO, potentially other DB-specific APIs
- May look at ORMs such as Doctrine with custom query languages

Discussion

Thank you!
Any Questions?

- Rascal: http://www.rascal-mpl.org
- Me: http://www.cs.ecu.edu/hillsma