CodeStats: Big Stats for Big Code

JOHN TOMAN and NATHANIEL YAZDANI

We present a software system, *CodeStats*, to empirically validate the prevalence of syntactic patterns in real-world codebases. Our prototype implementation supports only Java codebases. A domain-specific language for queries facilitates concise expression of syntactic patterns, while also mitigating misuse of the CodeStats computational infrastructure. For scalability, a distributed system executes the queries in map-reduce fashion. Our evaluation demonstrates that CodeStats can efficiently quantify the frequency of syntactic patterns useful to designers of program analyses across a significant amount of real-world source code.

1 INTRODUCTION

Designers of program analyses routinely trade off the accuracy (*soundness*) of their analysis in order to gain precision on program behaviors that matter most for real-world software. This practice permits a program analysis to make simplifying assumptions about difficult-to-model language constructs, such as self-interpretation (*e.g.*, JavaScript's `eval`). This practice has gained such momentum and acceptance that the term *soundy* has arisen in recent years to describe such program analyses [1, 2].

Unfortunately, designers of program analyses can only justify the profitability of such trade-offs by appealing to shared assumptions and intuitions among their own community of programmers, who are predominantly academic researchers. These assumptions and intuitions are not always accurate with respect to the reality of professional software development. For instance, program analyses for JavaScript effectively ignore self-interpretation, yet a recent study discovered that nearly all JavaScript programs make use of self-interpretation at some point, by fault of common libraries [3].

We present a system, *CodeStats*[1], with which the designer of a program analysis may (in)validate assumptions and intuitions about the empirical use of a language construct in real-world code.

2 OVERVIEW

The CodeStats system consists of three major design elements: the query language (section 3), the query engine (section 4), and the sample base of source code (section 5). The query language constrains the expressible queries to those deemed most useful for the intended purpose of CodeStats while also preventing undesirable or unintended uses of the computational infrastructure. The query engine efficiently evaluates a query across the sample base of source code, aggregates the results, and reports the final statistics. The sample base of source code serves as a representative set of source code with respect to which the frequency of the syntactic pattern described by a query is evaluated.

Currently, CodeStats is an early research prototype. While the system is not inherently specific to any one programming language, our prototype implementation is specialized

[1] https://github.com/uwplse/codestats

to Java, and the current design of the query language is somewhat specialized to object-oriented languages (*i.e.*, also applicable to languages like C# or Kotlin). One uses the system by uploading a query in our domain-speific language, which the query engine then evaluates across a sample set of codebases. The system outputs the calculated statistics along with some performance measurements to a log file.

3 QUERY LANGUAGE

The design of the query language promotes concision and mitigates misuse of the query engine. The query language is quite intentionally *not* Turing-complete; the language lacks recursion (*i.e.*, loops) and user-defined functions. A program in the query language can only assert a simple predicate on either expressions or statements and select how to count successful matches (*i.e.*, occurrences where the predicate is satisfied). The only recurrence possible in a query is due to the special indices `*` and `?`, which mean "for every element in the list" and "for at least one element in the list," respectively. This constrained expressivity prevents malicious or otherwise abusive users from exploiting the query engine for arbitrary computation. Figure 1 presents the syntax of the query language.

$$
\begin{aligned}
\textit{Program} &::= \textit{Query}^* \\
\textit{Query} &::= \textit{Ident}\ \texttt{:}\ \textit{Mode}\ \textit{Syntax}\ \textit{Ident}\ (\texttt{within}\ \textit{Ident})?\ \texttt{where}\ \textit{Conjunctive} \\
\textit{Mode} &::= \texttt{exists} \mid \texttt{count} \\
\textit{Syntax} &::= \texttt{expression} \mid \texttt{statement} \\
\textit{Conjunctive} &::= \textit{Disjunctive}\ (\texttt{and}\ \textit{Disjunctive})^* \\
\textit{Disjunctive} &::= \textit{Atomic}\ (\texttt{or}\ \textit{Atomic})^* \\
\textit{Atomic} &::= \textit{Attribute}\ \texttt{is}\ (\texttt{not})?\ (\texttt{constant} \mid \texttt{static} \mid \texttt{local} \mid \texttt{null}) \\
&\mid \textit{Attribute}\ \texttt{has type}\ \textit{String} \\
&\mid \textit{Attribute}\ \textit{Compare}\ \textit{Literal} \\
&\mid \texttt{(}\ \textit{Conjunctive}\ \texttt{)} \\
\textit{Attribute} &::= \textit{Ident}(\texttt{.}\textit{Ident}(\texttt{[}\textit{Index}\texttt{]})?)^* \\
\textit{Index} &::= \texttt{*} \mid \texttt{?} \mid \texttt{0} \mid \texttt{1} \mid \ldots \\
\textit{Compare} &::= \texttt{!=} \mid \texttt{==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \\
\textit{Literal} &::= \textit{Number} \mid \textit{String}
\end{aligned}
$$

Fig. 1. Syntax of the query language.

Query Definitions. A query in our domain-specific language is a sequence of *query definitions*, each of which names and describes an individual statistic to compute. The query language only supports predicated counting statistics, which tally the syntactic entities satisfying a predicate. The language supports predicates over two sorts of syntactic entities: expressions and statements. The `count` mode indicates a direct count of successful matches. For a bit greater flexibility, the `exists` mode limits the number of matches per

method to one; in other words, the `exists` mode enacts a tally of methods that contain at least one expression/statement satisfying the predicate. Lastly, a query definition may also refine another query definition using a `within` clause; the semantic meaning is as if the the refined query definition's predicate were conjoined before the current one's predicate.

Query Predicates. A *query predicate* is a collection of *atomic predicates* combined together with Boolean conjunction and disjunction. An atomic predicate asserts a simple property on *attributes* of the expression or statement. Depending on the kind of expression or statement, valid attributes may include the kind of expression or statement, the list of argument/operand expressions, the receiver expression, *etc.* *Trait checks* (*i.e.*, checks for `constant`, `static`, `local`, and `null`) are supported by expressions of any kind.

Attributes. Table 1 briefly summarizes the supported attributes in our prototype implementation of CodeStats. Essentially, attributes expose the interface of the Java abstract syntax tree (AST) rooted at the expression or statement. Since different expressions/statements may have different AST structure, accessing an invalid attribute falsifies the atomic predicate. The attributes `kind` and `host` are present on expressions and statements of any kind. To allow syntactic patterns that depend on the particular kind of expression or statement, query predicates are always evaluated with Boolean short-circuit semantics.

kind	supported attribute keys
unop	type, operand
array_ref	type, index, array
binop	type, operands, lop, rop
cast_expr	type, cast_type, castee
alloc	type, allocType, constrArgs
new_array	type, size, baseType
method_call	type, args, method
instance_method_call	type, args, method, receiver
fieldref	type, field
instance_fieldref	type, field, base_ptr
field	type, name, declaringClass
return_stmt	ret_val
assign_stmt	lhs, rhs
invoke_stmt	method_call
method	returnType, paramType, name, signature, declaringClass

Table 1. Summary of supported attributes.

Example. Figure 2 presents an example of a program in the query language. The statistic counts every expression that is an instance method invocation, in the body of a non-static method, passing the current receiver object for at least one formal argument. The identifier `self_argument` names the resultant statistic. This example demonstrates a common idiom in query definitions; earlier conjuncts progressively constrain the expression to the structure expected by the syntactic pattern, namely instance method invocations in non-static method bodies. This idiom prevents accidental falsification of the predicate due

to accessing an invalid attribute (recall that the set of valid attributes depends on the kind of expression or statement).

```
self_argument : count expression e where {
    e.kind == "InstanceInvoke" and
    e.host is not static and
    e.args[?] is this
}
```

Fig. 2. Example of a query in the domain-specific language.

4 QUERY ENGINE

The query engine is a distributed system consisting of three major components: the *query compiler*, the *query evaluator*, and the *query cluster*. The following paragraphs describe these components in turn.

Query Compiler. The query compiler translates a program in the query language (section 3) into a Java module. Attribute accesses are translated into field accesses and method calls on a polymorphic runtime representation of the AST for the current expression or statement. This representation permits flexibility, as the query compiler may add support for new attributes without significant (or possibly any) modification to the query language. For instance, adding support for a new language (*e.g.*, Scala) would not require any changes to the query language.

Query Evaluator. The query evaluator constructs the appropriate runtime representation for an expression or statement and calls the compiled Java module to perform the query. The query evaluator is built with the Soot framework for syntactic processing of Java bytecode. The query evaluator includes both a local and distributed runner, the former for debugging the system during development and the latter for use with the query cluster in a production (or rigorous testing) environment.

Query Cluster. The query cluster distributes the work to perform a query on the entire set of codebase samples over a pool of worker nodes in map-reduce fashion. Since the query language constrains user queries to be simple, local, and fine-grained, the overall work to perform a user query is massively parallelizable; a distributed architecture can exploit this property to scale the query engine to a potentially massive sample of codebases ("big code"). To achieve this, the query cluster is built on top of the Hadoop framework for distributed data processing. The mapper simply invokes the query evaluator over a chunk of the dataset (*i.e.*, a chunk of Java bytecode), and the reducer aggregates the results of query evaluations as specified by the user's query. For resiliency, the mapper can tolerate local failures from the query evaluator, which may occur due to bugs in the query compiler, partially corrupted bytecode, or bytecode from an unsupported version of Java. The query cluster pulls the data (*i.e.*, sample source code) from a cache stored on an HDFS volume.

5 CODEBASE SAMPLES

For the sake of our prototype system, we manually collected and prepared sample codebases for use by the CodeStats system. We drew about half of our codebase samples from the GitHub Trending list for Java repositories; for the rest, we used significant, well-known Java projects, such as Eclipse and several Apache projects. To prepare the sources for use by CodeStats, we compiled each project into Java archives (`.jar` files), extracted the bytecode, and uploaded the bytecode into Hadoop stream files onto a Hadoop Distributed Filesystem (HDFS) connected to the query engine's cluster. Table 2 lists the codebases used as samples.

Name	Repository
Kotlin Compiler	https://github.com/JetBrains/kotlin/tree/master/compiler
LibreOffice	https://cgit.freedesktop.org/libreoffice
mvnForum	https://sourceforge.net/projects/mvnforum/files/mvnForum/
OkHttp	https://github.com/square/okhttp
Mozilla Rhino	https://github.com/mozilla/rhino
RxJava	https://github.com/ReactiveX/RxJava
Apache Solr	https://repo1.maven.org/maven2/org/apache/solr/
Spring Framework	https://github.com/spring-projects/spring-framework
Apache Tomcat	https://github.com/apache/tomcat
Apache Commons	https://repo1.maven.org/maven2/org/apache/commons/
Eclipse	https://git.eclipse.org/c/
Elasticsearch	https://github.com/elastic/elasticsearch
Glide	https://github.com/bumptech/glide
Groovy	https://github.com/apache/groovy
Google Guava	https://github.com/google/guava
H2 Database Engine	https://github.com/h2database/h2database
Javalin	https://github.com/tipsy/javalin
Jenkins	https://github.com/jenkinsci/jenkins
Jetty	https://github.com/eclipse/jetty.project

Table 2. Codebases used for sample source code.

We used this methodology so that our sample codebases were representative of professional-quality software, the usual target for program analyses. A more mature version of CodeStats could automatically crawl the web for Java sources to use, though this could introduce concerns around the selection criteria, due to the potential for undesired statistical bias in the sample codebases.

6 EVALUATION

We evaluated CodeStats against the project goal of validating simplifying, yet semantically significant, assumptions about program behavior used in the design of program analyses. For this evaluation, we chose to investigate the following such program behaviors:

(1) Accessing public fields on an object of a different class
(2) Using the `Object`-provided condition-variable mechanism

(3) Instantiating language-native arrays, especially with dynamic size
(4) Invoking language reflection with dynamic arguments

Assuming the absence of program behavior (1) is useful for program analyses based on separation logics, in which the heap is partitioned into disjoint, non-interfering regions. Such an assumption would be reasonable because the common practice in object-oriented languages like Java is to provide getters and setters for fields that validate all updates.

Assuming the absence of program behavior (2) is useful for program analyses simply due to its complexity, which necessitates concurrent reasoning. Such an assumption is reasonable because the `Object`-provided condition-variable mechanism is widely believed to be uncommonly used.

Assuming the absence of program behavior (3) is useful because array reasoning is prone to permeate the design of a program analyses[2]. Such an assumption is reasonable because using higher-level collection abstractions (*e.g.*, lists) is widely considered better programming practice, as they offer greater convenience.

Assuming the absence of program behavior (4) is useful due to the intractability of modeling arbitrary invocations of language reflection. Such an assumption is reasonable — at least for the evaluation of our system — because it is standard among program analyses for Java.

Results. We implemented queries for CodeStats to quantify the prevalence of the previously mentioned program behaviors, each of which admits a precise syntactic pattern in the query language. We hosted the query engine on a cluster of 3 dedicated virtual servers (m4.large configuration, with 2 virtual processors and 8 GiB of memory) on the Amazon Web Services (AWS) cloud-computing platform, augmented by 3 comparable "spot" instances[3].

Table 3 presents the statistics computed to quantify the prevalence of the previously described program behaviors. Related statistics were grouped together into combined queries. Where informative, we performed method counts (`exists` statistics) in additional to the raw expression counts (`count` statistics). The codebase sample set, described in greater detail by section 5, consisted of 1,024,351 methods and approximately 200 times as many expressions (as counted by bytecode instructions).

[2]Moreover, higher-level collections may admit simpler, more effective semantic models in a program analysis.
[3]An AWS "spot instance" is a transient virtual server that consumes spare computational resources in the datacenter.

Description	Total occurrences	Methods with 1+ occurrences	Query time
Read of public field	134,275	36,939	25m, 46s
Write to public field	29,315	9,749	
Condition-variable call	1,206	987	25m, 36s
Array creation	101,913	-	25m, 32s
Array creation (constant)	80,208	-	
Reflection call	3,507	-	25m, 36s
Reflection call (dynamic)	1,309	-	

Table 3. Statistics computed for evaluation.

Analysis. Our results validate and invalidate some common assumptions about the chosen program behaviors. As expected, accesses to public fields, particularly (and most critically) for writes. Similarly, we found that use of the `Object`-provided mechanism for condition variables is exceedingly rare. Interestingly, we found that the majority of array instantiations, while somewhat rare, used constant sizes; this could make a useful assumption in the design of a program analysis. Contrary to the widely held belief by the research community, we found that use of language reflection more commonly involves dynamic arguments. (While uncommon, the few uses of language reflection typically serve a critical function in the overall architecture of a Java framework.)

7 DISCUSSION

Long term, our vision is to develop a public web service. Through this web service, the designer of a program analysis may submit a query to validate an assumption about the prevalence of a syntactic pattern and receive a publicly accessible link to a quantitative report of the systems' findings. When the designer publishes a paper on their program analysis, they may reference this report to justify their assumptions. Not only could this assuage concerns over more common assumptions, it could enable designers to make further useful assumptions about language features, even when the infrequency of the corresponding syntactic pattern goes against the intuition of the research community. Realizing this vision requires a significant investment of time and money, neither of which is currently available for the project in the scope of one quarter.

8 CONCLUSION

This report presented CodeStats, a system to calculate the frequency of syntactic patterns in a representative sample base of Java source code. The target application is for designers of program analyses to validate their assumptions about the prevalence of difficult-to-model language constructs. A domain-specific query language provides a concise interface that also guards against misuse of the system, and a distributed query engine efficiently evaluates user queries at scale. The report's evaluation supported the efficacy of CodeStats for its intended application on a large sample set of real-world codebases.

REFERENCES

[1] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 332–343. DOI: http://dx.doi.org/10.1145/2970276.2970347

[2] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. DOI: http://dx.doi.org/10.1145/2644805

[3] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 52–78. http://dl.acm.org/citation.cfm?id=2032497.2032503