Programming Language Evolution via Source Code Query Languages

Raoul-Gabriel Urma
Computer Laboratory, University of Cambridge
raoul.urma@cl.cam.ac.uk

Alan Mycroft
Computer Laboratory, University of Cambridge
alan.mycroft@cl.cam.ac.uk

Abstract

Programming languages evolve just like programs. Language features are added and removed, for example when programs using them are shown to be error-prone. When language features are modified, deprecated, removed or even deemed unsuitable for the project at hand, it is necessary to analyse programs to identify occurrences to refactor.

Source code *query languages* in principle provide a good way to perform this analysis by exploring codebases. Such languages are often used to identify code to refactor, bugs to fix or simply to understand a system better.

This paper evaluates seven Java source code query languages: Java Tools Language, Browse-By-Query, SOUL, JQuery, .QL, Jackpot and PMD as to their power at expressing queries required by several use cases (such as code idioms to be refactored).

Categories and Subject Descriptors D.2.0 [*Software Engineering*]; D.3.0 [*Programming Languages*]

General Terms Languages

Keywords program analysis, source code, query languages

1. Introduction

Programming languages evolve just like programs. Language features are added and removed, for example when programs using them are shown to be error-prone.

Language designers often make use of program analysis to understand the impact of language features. They identify locations of a specific feature in a corpus of programs to learn whether it should influence the evolution of the language [12]. In addition, when language features are modified, deprecated, removed or even deemed unsuitable for the project at hand, it is necessary to analyse programs to identify occurrences to refactor.

Various automated tools have been developed to assist programmers analyse their programs. For example, several code browsers have been developed to help program comprehension through hyperlinked code and simple queries.

Recently, source code query languages have been developed to provide extensive analysis of source code. They let programmers compose queries written in a domain specific language to locate potential bugs, code to refactor or simply to understand a system better. Many query languages have been developed with different characteristics and features. For example, some source code query languages are based on *Abstract Syntax Tree* expressions, some provide an *SQL-like* feel, others include features from *logic programming* to build more complex queries.

In principle, source code query languages could help language designers perform program analysis to learn whether a feature is prevalent enough to influence the evolution of the language. In fact, a recent empirical study made use of a source code query language to investigate the use of overloading in Java programs [11].

In this paper, we compare seven recent source code query languages: *Java Tools Language*, *Browse-By-Query*, *SOUL*, *JQuery*, *.QL*, *Jackpot*, *PMD* [1, 4, 6, 8–10, 17] and evaluate whether their expressiveness suffices for language design research. To this end, we study several use cases of recent Java features and their design issues – investigating their expressibility as queries in the languages we examine.

2. Java Source Code Query Languages

This section gives an overview of the seven source code query languages we evaluate: *Java Tools Language*, *Browse-By-Query*, *SOUL*, *JQuery*, *.QL*, *Jackpot* and *PMD*. We selected these languages because they provide a variety of design choices and let the programmer specify user-defined queries. For example, we excluded *Findbugs* as it only lets programmers query source by creating new classes based on a Java framework [3]. We also only selected source code query languages that include a guide or a working implementation. Figure 1 gives a brief summary.

Name	Paradigm	Query: "find all methods named `bar`"	Input	Date	Licence
JTL	Logic	`public bar(*)`	Bytecode	2006	Open Source
BBQ	Declarative	`matching "bar" methods in all classes`	Bytecode	2005	Open Source
SOUL	Logic	`if jtMethodDeclaration(?m) {` `public ?type bar(?paramList) { ?statements };` `}`	Source	2011	Open Source
JQuery	Logic	`method(?M,name,bar)`	Source	2003	Open Source
.QL	OO, SQL-like	`From Method m where m.hasName("bar") select m`	Source	2007	Commercial
Jackpot	Declarative	`$modifiers$ $returnType foo($args$)` `throws $thrown$ {` `$bodyStatements$;` `}`	Source	2009	Open Source
PMD XPath	XPath	`//MethodDeclarator[@Image = "bar"]`	Source	2004	Open Source

Figure 1. Overview of the analysed Java source code query languages

Java Tools Language (JTL) is a logic-paradigm query language to select Java elements in a code base and compose data-flow queries [8]. The implementation analyses Java bytecode classes. JTL syntax is inspired by Query-by-Example [17]. It also provides *variable binding*, which allows queries to capture a match in a variable. This allows queries to refer back to a match (e.g. to find two method declarations with the same name and declared in the same class).

Browse-By-Query (BBQ) reads Java bytecode files and creates a database representing classes, method calls, fields, field references, string constants and string constant references [1]. This database can then be interrogated through English-like queries. In addition, BBQ allows more complex queries via filtering mechanisms and set operators.

SOUL is a logic-paradigm query language [10]. It contains an extensive predicate library called *CAVA* that matches queries against AST nodes of a Java program generated by the *Eclipse JDT*. SOUL uses a combination of logic queries and template matching of a Java code. In practice, this means a user can create a logic variable to match an AST node and reuse this variable within the query regardless of the execution path where the variable appears.

JQuery is a logic-paradigm query language built on top of the logic programming language *TyRuBa* [13]. It analyses the AST of a Java program by making calls to the Eclipse JDT. JQuery includes a library of predicates that lets the user query Java elements and their relationships.

.QL is an object-oriented query language. It enables programmers to query Java source code with queries that resemble SQL [9]. This design choice is motivated as reducing the learning curve for developers. In addition, the authors argue that object-orientation provides the structure necessary for building reusable queries. A commercial implementation is available, called *SemmleCode*, which includes an editor and various code transformation.

Jackpot is a module for the *NetBeans IDE* for querying and transforming Java source files [4]. Jackpot lets the user query the AST of a Java program by means of a template representing the source code to match. It supports variable binding within queries.

PMD is a Java source code analyser that identifies bugs or potential anomalies including dead code, duplicated code or overcomplicated expressions [6]. It has an extensive archive of built-in rules that can be used to identify such code. One can specify new rules by writing them in Java and making use of the PMD helper classes. Alternatively, one can also compose custom rules via an *XPath* expression that queries the AST of the program. As we concentrate on source code queries, we only evaluate the XPath facilities of PMD.

3. Use Cases

In this section, we describe the use cases we chose; these highlight current language design discussions in the research community and by language designers as well as exploring various Java features.

3.1 Generic Constructors

A constructor has two sources of type arguments. It can use the type parameters declared in a generic class. Additionally, it can declare its own type parameters. The code below illustrates a constructor of class `Foo` which declares its own type parameter `S` that extends the class's parameter.

```
class Foo<T extends Number> {
    <S extends T> Foo() {}
}
```

The combination of the diamond operator (`<>`, introduced in Java 7, which allows inference of generics arguments) together with explicit constructor type arguments was banned as it caused problem with the type inference mechanisms. In addition, the language designers argued that this restriction has little impact because generic constructors are rare in practice [2].

Use Case 1: Find generic constructors whose type parameters extend the enclosing class's own type parameters.

3.2 Overloaded Methods

Overloading methods allows programmers to declare methods with the same name but with different signatures.

For example, one could write an add method that takes a different number of parameters:

```
public void add(T a) { ... }
public void add(T a, T b) { ... }
```

Often this pattern can be refactored using *varargs* if the overloaded methods' parameters share a single type:

```
public void add(T a, T ... args) { ... }
```

Related to this use case, recent work has investigated overloading in Java and found that a quarter of overloaded methods are simulating default arguments. The authors used the source code query language *JTL* to query a corpus for overloaded methods [11].
Use Case 2: Find overloaded methods with multiple parameters that share a single type.

3.3 Covariant Arrays

In Java and C#, array subtyping is *covariant*, meaning that type `B[]` is considered a subtype of `A[]` whenever `B` is a subtype of `A`. However, this relation can cause run-time exceptions [15]. Consider the following Java code where `Banana` and `Apple` are subtypes of `Fruit`:

```
Banana[] bananas = new Banana[5];
Fruit[] fruit = bananas;
fruits[0] = new Apple(); // ArrayStore Exception
peelBanana(bananas[0]);  // Apple???
```

The assignment to `fruit[0]` on line 3 will cause an ArrayStore exception. Although statically, the variable `fruit` has type `Fruit[]`, its run-time type is `Banana[]` and thus we cannot use it to store an `Apple`. We recently conducted a corpus analysis to investigate the use of covariant arrays. We found that these are rarely used in practice and can be refactored using generics [16].
Use Case 3: Find occurrences of covariant array uses in assignment, method calls, constructor instantiations and return statements.

3.4 Rethrown Exceptions

Java 7 introduced improved checking for rethrown exceptions. Previously, a rethrown exception was treated as throwing the type of the `catch` parameter. Now, when a `catch` parameter is declared or effectively `final`, the type is known to be only the exception types that were thrown in the `try` block and are a subtype of the `catch` parameter type.

This new feature introduced two source incompatibilities with respect to Java 6. In practice, the same program will therefore have a different meaning if compiled with the Java 6 and Java 7 compiler. This exemplifies the notion of *quiet changes* in the ANSI/ISO C rationale document. Quiet changes are discouraged because existing programs may behave differently without warning.

Related to this use case, the Java language designers conducted a corpus analysis to investigate nested `try/catch` blocks and understand the impact of this change on users [12].
Use Case 4: Find occurrences of nested `try/catch` blocks that rethrow an exception.

4. Evaluation

This section reports our evaluation of the source code query languages and whether they could support our use cases.

Java Tools Language We did not find a working Eclipse plug-in for JTL. Our evaluation is based on the available documentation of JTL. We found that the use cases could not be specified because JTL lacks support for statements, generics and types of expressions. Since JTL is based on bytecode analysis rather than source code analysis, several queries are inherently restricted. For example, Java generics are compiled to casts and macro-style constructs such as `foreach` can be *desugared* during the bytecode translation.

Browse-By-Query We found that all except the overloaded-method use case could not be specified in BBQ. It lacks support for statements, generics and types of expressions.

SOUL We found that all our use cases could be specified in SOUL. It supports all the constructs available in Java including local variable declarations, generics and control flow statements. SOUL also provides built-in access to the Eclipse semantic analyser to retrieve types of expressions. This enables the covariant arrays use case to be queried. However, because SOUL relies on the Eclipse API some queries have accidental complexity and become verbose.

To tackle this issue SOUL provides a template matching mechanism: a query can represent a template of the Java code that needs to be matched. However, not all Java constructs are currently supported. For example, there is no support for array accesses, generics, `try/catch` statements. Hence our use cases cannot be specified using solely the template matching feature of SOUL.

JQuery We found that JQuery could specify none of our use cases as it does not support local variables, statements, generics and types of expressions. Nonetheless, JQuery supports variable binding and predicates on method declarations, which can express queries for overloaded methods. However, it cannot express the additional constraint that the overloaded methods need to share a single type.

.QL .QL is a commercial product and no reference material is available. Our evaluation is based on private communication with the authors. .QL supports all language features in Java as well as provides types of expressions. In addition, .QL supports variable binding within queries. This combination matches the expressivity of SOUL. While the query language syntax is not yet public, the authors confirmed that it can express all our use cases.

Jackpot We found that Jackpot cannot specify two use cases. Firstly, the covariant array use case is not supported because Jackpot does not provide any type information for AST nodes. Also the overloaded-method use case could not be specified because the matching engine does not support such queries. However, the generic constructors and rethrown-exception use cases could be specified using the template-matching and variable features of Jackpot.

PMD XPath We found that PMD supports querying of all Java constructs. In fact the entire program AST is stored in XML format. However, as PMD lacks a variable-binding mechanism within a query, it is not possible to refer back to a match. In addition, the types of expressions are not available. Hence none of our use cases could be specified.

5. Discussion and related work

We found that the source code query languages which could express all our use cases shared four characteristics: *complete support of AST*, *attributed AST nodes*, *variable binding* and *filtering* mechanisms.

Adding information from other forms of program analysis to query languages could ease the programming language evolution. For example, a query language could provide support for querying the class inheritance graph of a source code in order to perform a class hierarchy analysis. Similarly, it could access dataflow information to express whether a variable is live or the number of objects it may point to.

In addition, exposing the wider structure and state generated by a compiler could give further useful insights. For example, with access to inference analysis, a source code query language could provide syntax to test the impact of different type inference rules. This is a current issue of discussion for Java 8 which has more aggressive type inference [5]. Moreover, making run-time information available could be useful in order to investigate the impact of a feature at run time such as its memory consumption. In fact, existing query technologies can analyse a Java heap to facilitate application troubleshooting.

We also believe that improved reporting could provide researchers better insights to evaluate the results of a query. For example, recent work analysed the use of generics and their adoption [14]. A query that finds occurrences of generics is not as useful as reporting their instances along with contextual information when finding patterns of use.

Alves et al. described a comparative study of code query technologies [7]. They specified a software metric query in each language and compared it against twelve criteria such as *modularity*, *output format* and *licensing*. Our work differs because we evaluate whether source code query languages can help answer programming language design questions.

6. Conclusion

We researched whether source code query languages can help programming language evolution, evaluating whether seven such query languages can query for code idioms and recent language design issues in Java. We found that only *SOUL* and *.QL* provide the minimal features required to express all our use cases: namely variable binding, filtering mechanisms, complete AST and attributed AST nodes. We also suggested possible extensions to query languages, such as data-flow queries, to enhance their search capability.

Acknowledgments

We thank Alex Buckley, Joel Borggrén-Franck, Maurizio Cimadamore, Jan Lahoda, Oege de Moor and Coen De Roover for helpful comments. This work was supported by a Qualcomm PhD studentship and an Oracle work placement.

References

[1] BBQ. `http://browsebyquery.sourceforge.net/`.

[2] Diamond and generic constructors. `https://blogs.oracle.com/darcy/entry/project_coin_diamond_generic_constructors`.

[3] Findbugs. `http://findbugs.sourceforge.net`.

[4] Jackpot. `http://wiki.netbeans.org/Jackpot`.

[5] Java 8: support for more aggressive type-inference. `http://mail.openjdk.java.net/pipermail/lambda-dev/2012-August/005357.html`.

[6] PMD. `http://pmd.sourceforge.net/`.

[7] T. Alves, J. Hage, and P. Rademaker. A comparative study of code query technologies. In *SCAM*, 2011.

[8] T. Cohen, J. Y. Gil, and I. Maman. JTL: The Java tools language. In *OOPSLA*, 2006.

[9] O. de Moor, M. Verbaere, and E. Hajiyev. Keynote address: .QL for source code analysis. In *SCAM*, 2007.

[10] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ*, 2011.

[11] J. Gil and K. Lenz. The use of overloading in Java programs. In *ECOOP*, 2010.

[12] B. Goetz. Language designer's notebook: Quantitative language design. `http://www.ibm.com/developerworks/java/library/j-ldn1/`.

[13] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD*, 2003.

[14] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Mining Software Repositories*, 2011.

[15] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[16] R.-G. Urma and J. Voigt. Using the OpenJDK to investigate covariance in Java. *Oracle Java Magazine*, 2012.

[17] M. M. Zloof. Query by example. In *AFIPS National Computer Conference*, pages 431–438, 1975.