

CodeQuest: Querying Source Code with DataLog

Elnar Hajiyev¹, Mathieu Verbaere¹, Oege de Moor¹ and Kris de Volder²

¹ Programming Tools Group
University of Oxford
United Kingdom

² Software Practices Lab
University of British Columbia
Vancouver, Canada

elmar.hajiyev@comlab.ox.ac.uk,
mathieu.verbaere@comlab.ox.ac.uk, kdvolder@cs.ubc.ca
oege@comlab.ox.ac.uk

ABSTRACT

We describe *CodeQuest*, a system for querying source code. It combines two previous proposals, namely the use of logic programming and database system. Experiments (on projects ranging from 3KSLOC to 1300KSLOC) confirm that for this application, a query language based on DataLog strikes the right balance between expressiveness and scalability.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments;
H.3.4 [Information Storage and Retrieval]: Systems
and Software—*source code querying*

General Terms

Languages, Design, Measurement, Performance

Keywords

Source code querying, analysis of object-oriented programs,
DataLog, relational databases

1. INTRODUCTION

A *code query* helps to identify locations of interest in the source of a program. The most commonly used tool for code queries is *grep*, but often it is desirable to ask semantical questions that cannot be solved by purely text-based pattern matching. Such semantic code queries are important for checking coding styles (such as naming conventions based on types), fault detection (to discover bugs at development time), refactoring (to detect code smells), and aspect weaving (to identify join point shadows of interest). Perhaps the most common use, however, is simply for program understanding.

In several of these application areas, *e.g.* aspect weaving, there have been numerous proposals for specialised query languages. These range from complex pattern languages (as in AspectJ [7]) to Prolog as part of LogicAJ [4]. Tools that are intended for interactive use in a development environment exhibit a similarly wide range of query technologies, ranging from a generalisation of AspectJ's pointcuts in the CME [9] to a full logic programming language in JQuery [5].

In the software maintenance community, there is a long tradition of using storing information about the source in a

database, and then querying that information via database queries, *e.g.* [1, 8]. Jarzabek [6] presents an SQL-like query language with special primitives for code queries, implemented on top of a Prolog system. ASTLog is a query language for inspecting abstract syntax trees [2].

Thus far there has not been a rigorous assessment of the relative merits of these different technologies. In particular, there appears to be little work on assessing scalability; and yet the importance of code queries grows with program size. This poster represents the first steps towards such an assessment, as well as a synthesis of the best ideas of the works cited above. To wit, the contributions of the work reported here are:

- The combination of earlier strands of work, leading to the use of DataLog as the query language, with a traditional database system as its backend.
- The construction of an optimising compiler from DataLog to SQL, both targeting built-in recursive queries and a custom implementation of recursion via stored procedures.
- The collection of a number of benchmarks to compare query engines.
- A comparison of four different query engine technologies with respect to these benchmarks.

2. EXAMPLE CODE QUERIES

We now present some example queries to help focus the discussion. All three queries are intended to run on Java programs. The notation we use is that of Prolog, as suggested by many of the works cited above.

The first query is checking a common style rule, namely that there are no declarations of non-final public fields. When such fields occur, we want to return both the field F and the enclosing type T . As a Prolog clause, this query might read as follows:

$$q_1(T, F) \text{ :- } \text{type}(T), \text{child}(T, F), \text{field}(F), \\ \text{modifier}(F, \text{public}), \\ \text{not}(\text{modifier}(F, \text{final})) .$$

Our second query is a little more interesting. Here we wish to determine all methods M that write a field of a particular type, say T . In fact, fields whose type is a *subtype* of T qualify as well. We therefore specify:

$$q_2(M, T) \text{ :- } \text{method}(M), \text{writes}(M, F), \\ \text{type}(F, FT), \text{subtypestar}(T, FT) .$$

Copyright is held by the author/owner.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

Here the main relation of interest is $subtypestar(T, FT)$, which relates a type T to its subtypes FT . It is defined recursively:

$$\begin{aligned} subtypestar(T, T) & :- type(T) . \\ subtypestar(T, FT) & :- subtype(T, V), subtypestar(V, FT) . \end{aligned}$$

In turn, the *subtype* relation is defined in the obvious way, in terms of the *implements* and *extends* relations. A number of other researchers (e.g. [6]) have proposed special primitive relations to avoid the use of explicit recursion. In our opinion, recursion is essential to keep the language simple while achieving the desired expressive power.

Our final query is to find all implementations M_2 of an abstract method M_1 :

$$q_3(M_1, M_2) :- \begin{aligned} & modifier(M_1, abstract), \\ & overrides(M_2, M_1), \\ & \text{not}(modifier(M_2, abstract)) . \end{aligned}$$

The definition of *overrides* does of course make use of *subtypestar*; we omit details.

3. DATALOG

While the descriptions in the previous section are certainly attractive, the use of logic programming has a number of problems. First, all facts must be kept in memory, and therefore it does not meet the scalability criterion. Furthermore, to achieve the desired efficiency even on modest examples, it is necessary to pollute the elegant specifications with mode annotations (saying which variables are input and which are output), and the use of the cut operator to control backtracking.

Fortunately it turns out that typical code queries do not require the full power of Prolog. In particular, it is rare for code queries to build up intermediate data structures such as lists. We can therefore restrict our attention to the *DataLog* language [3], which is essentially Prolog without data structures: it manipulates relations only. This subset is expressive enough for code queries, and it is furthermore possible to provide a scalable implementation on top of existing database systems.

4. EXPERIMENTS

We compared four different query engines on four different projects, for the queries presented earlier. The query engines are: JQuery [5]; XSB, an optimising compiler for Prolog with tabling; Microsoft SQL Server with built-in recursion via CTEs; and Microsoft SQL Server with our own implementation of recursions via stored procedures. We chose a tabled implementation of Prolog because without tabling, Prolog is not at all competitive with the other technologies.

The projects we ran the queries on are Jakarta (3 KSLOC), the source of JQuery (38 KSLOC), JFreeChart (93KSLOC) and Eclipse (1311 KSLOC). These projects were chosen because of their wide availability, as well as their size characteristics. The benchmark results are shown in Figure 1. For Eclipse, XSB starts seriously lacking performance. Although it is not shown in the figure, the compilation of the facts requires more than 700MB of RAM and lasts over 10 minutes. As for JQuery, there are no bars for Eclipse since it was not possible to parse the project entirely.

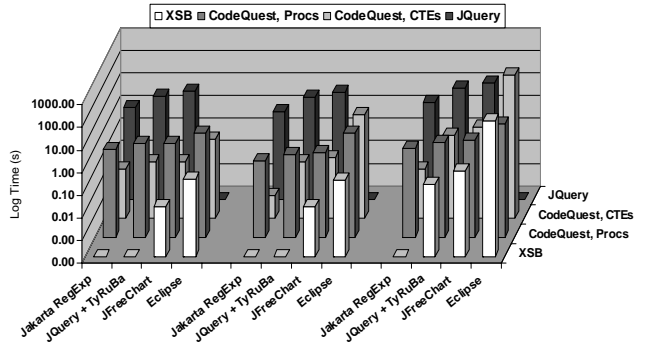


Figure 1: Benchmark results for q_1 , q_2 and q_3 .

5. CONCLUSIONS

We have started to compare different technologies for implementing code query engines. It is clear that for small code bases, a tabled implementation of Prolog such as XSB performs extremely well. For scalability, however, an implementation of DataLog on top of a database system is much more preferable. We are currently verifying these conclusions via a system that logs code queries while performing whole tasks: this will thus provide better benchmark queries.

In future work, we also plan to investigate yet more aggressive optimisations of recursion - as it stands, we do considerably better than the built-in facilities of Microsoft SQL Server. We plan to integrate our system with an interactive development environment such as Visual Studio. An important problem is then to update the database incrementally, using the recompilation manager of the IDE. Finally, we plan to integrate this technology with the AspectBench Compiler for AspectJ.

6. REFERENCES

- [1] Y. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.
- [2] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.
- [3] H. Gallaire and J. Minker. *Logic and Databases*. Plenum Press, New York, 1978.
- [4] Stefan Hanenberg, Günter Kniessel, and Tobias Rho. Evolvable pattern implementations need generic aspects. In *Proc. of ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*. June 2004.
- [5] Doug Janzen and Kris de Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.
- [6] Stan Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215, 1998.
- [7] Gregor Kiczales, John Lamping, Anurag Menhdekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
- [8] Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.
- [9] Peri Tarr, William Harrison, and Harold Ossher. Pervasive query support in the concern manipulation environment. Technical Report RC23343, IBM Research Division, Thomas J. Watson Research Center, 2004.