MetaDL: Declarative Program Analysis for the Masses

Alexandru Dura
Dept. of Computer Science
Lund University
Lund, Sweden
alexandru.dura@cs.lth.se

Hampus Balldin
Dept. of Computer Science
Lund University
Lund, Sweden

Abstract

While Datalog provides a high-level language for expressing static program analyses, it depends on external tooling to extract the input facts from the analyzed programs. To remove this dependency, we present MetaDL. The MetaDL system consists of a Datalog language extension for source-level program analysis and tools for generating the language extension from a description of the analyzed language.

CCS Concepts • **Software and its engineering → Automated static analysis**; **Constraint and logic languages**; *Domain specific languages*; • **Theory of computation →** *Pattern matching*.

Keywords Datalog, Domain-Specific Languages, Pattern Matching, Static Analysis

ACM Reference Format:
Alexandru Dura and Hampus Balldin. 2019. MetaDL: Declarative Program Analysis for the Masses. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '19), October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3359061.3362781

1 Introduction

In contrast to imperative implementations of program analyses, Datalog eliminates the need for manual worklist management and provides a higher-level notation to express the derivation of new facts, until a fixpoint is reached.

Frameworks that harness the expressive power of Datalog to implement complex points-to analyses have emerged, targeting the Java language [2] and also intermediate representations such as LLVM IR [1]. These frameworks consist of two major components: the fact extractor and the actual analysis. The fact extractor processes the input program and extracts the relevant information into a fact database. The analysis uses this database as input to its inference rules to compute the results. Typically, the Datalog program is evaluated by a high performance engine such as Soufflé[5], which enables the analysis to scale to large programs.

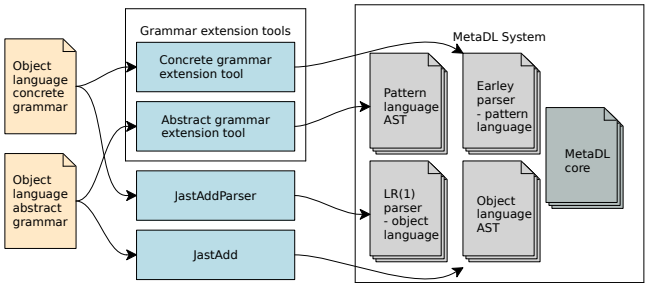

Figure 1. Build-time view of the MetaDL system

While the declarative nature of Datalog greatly simplifies the implementation of an analysis, developing a fact extractor for a specific language requires specialized knowledge and also demands a considerable engineering effort. These are acceptable constraints for the development of complex analyses, but they also hinder the adoption of declarative program analysis by software practitioners, for everyday tasks such as extraction of code metrics, automatic code reviewing or program understanding.

To leverage on the advent of high performance Datalog engines and to bring the advantages of declarative program analysis to the field, we propose the MetaDL system. The MetaDL system combines the expressivity of Datalog and the simplicity of syntactic patterns to allow users to reason about their programs using familiar tools: first order logic and the analyzed programming language itself.

2 The MetaDL System

The MetaDL system consists of the *MetaDL core language* and a set of *grammar extension tools*. The *MetaDL core language* is an extension of Datalog with syntactic patterns [3]. The syntactic patterns allow the user to match subtrees in the AST of the analyzed program and to refer to the matched subtrees in other plain Datalog atoms. To illustrate, the pattern

```
<: class `c implements .., `i, .. { .. } :>
```

expresses the "class `c` implements interface `i`" relation from a Java program. The language of the syntactic patterns extends the object language with *metavariables* (e.g. `` `c `` and `` `i ``) and *gaps* (`..`). *Metavariables* bind to any AST subtree that is allowed by the object language in their position. In our framework, list nodes represent sequences (e.g. implemented interfaces, statements, method parameters), while all other

SPLASH Companion '19, October 20–25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6992-3/19/10.
https://doi.org/10.1145/3359061.3362781

node kinds have a fixed number of children. *Gaps* represent ignored elements inside these list nodes.

By using a description of the concrete and abstract grammars, the *grammar extension tools* generate the language of the syntactic patterns. While we currently support only Java 1.4 and MetaDL itself, the grammar extension tools can be used to add new languages with minimal effort.

2.1 System Configuration

In Figure 1 we provide an overview of the configuration process of the MetaDL system for a given programming language. To instantiate the system for a particular language, the user must provide an EBNF description of the concrete grammar and a description of the abstract grammar.

To extend the abstract grammar with metavariables and gaps, we use a modified version of JastAdd [4], a meta-compilation framework. A related tool, JastAddParser[1], is used to generate the *pattern grammar*. The *pattern grammar* is a super-set of the *object grammar*, where every non-terminal has two supplementary productions: a metavariable and a gap. While an LR(1) parser is used for the object language and for the main MetaDL program, the pattern language is possibly ambiguous and thus requires the use of an Earley parser[2] that generates all the possible parse trees. The parsers and the classes describing the abstract grammars are packaged together with the MetaDL core language.

2.2 Runtime

At runtime (Figure 3), an instance of the MetaDL system takes as input a MetaDL file and one or multiple files containing the programs under analysis. MetaDL invokes the object language parser which produces the ASTs of the analyzed programs. The resulting ASTs are then transformed to program representation relations. In the same stage, MetaDL generates ASTs for the syntactic patterns and transforms these ASTs to plain Datalog clauses (cf. [3]). The execution continues with the translation of the resulting program into the Datalog dialect used by the Soufflé engine [5]. The program representation relations are also serialized. Finally, MetaDL invokes the Soufflé engine to compute the results.

2.3 The MetaDL Core Language

In Figure 2 we illustrate an instance of the MetaDL system for the Java language, with a program which collects the method arities. On line 1, we use the IMPORT pseudopredicate to declare that the P predicate represents the Java program in the file `example.java`. We continue with an **`analyze`** block, which establishes a context for the syntactic patterns, ID and INDEX pseudopredicates, which refer to the program represented by P. The ID pseudopredicate (lines 5, 8) maps an identifier to its name, while the INDEX pseudopredicate

[1] http://jastadd.org/web/tool-support/jastaddparser.php
[2] https://github.com/coffeeblack/pep

```
1  IMPORT('P, "example.java", "java4").
2  analyze('P, "java4") {
3    MethodDefiniton(cname, name, 0) :-
4      <: class `c { .. `t `m() { .. } .. } :>,
5      ID(`c, cname), ID(`m, name).
6    MethodDefiniton(cname, name, nargs) :-
7      <: class `c { .. `t `m(.., `arg) { .. } .. } :>,
8      ID(`c, cname), ID(`m, name),
9      INDEX(`arg, iarg), nargs := iarg + 1.
10 }
11 OUTPUT('MethodDefinition).
```

Figure 2. MetaDL-Java program that lists arities of methods

(line 9) maps an AST node which is a list element to its position. On lines 4 and 7, metavariables `` `c ``, `` `t `` and `` `m `` connect the syntactic patterns to the surrounding Datalog clause, while gaps (. .) mark ignored class-level declarations, statements and parameter declarations.

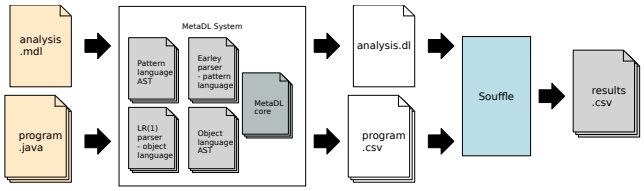

Figure 3. Runtime view of the MetaDL system

3 Future Work

In order to evaluate our system on modern applications, we aim to support Java 8 as an object language. In addition, we are investigating adding code transformation capabilities and also exposing JastAdd AST attributes as Datalog predicates.

Acknowledgments

This work was partially supported by Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

References

[1] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium*. Springer, 84–104.

[2] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of OOPSLA '09*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[3] Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. 2019. MetaDL: Analysing Datalog in Datalog. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2019)*. ACM, New York, NY, USA, 38–43. https://doi.org/10.1145/3315568.3329970

[4] Görel Hedin and Eva Magnusson. 2003. JastAdd: An Aspect-oriented Compiler Construction System. *Sci. Comput. Program.* 47, 1 (April 2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0

[5] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th Int. Conf. on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226