MetaDL: Analysing Datalog in Datalog

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

Christoph Reichenbach
Dept. of Computer Science
Lund University
Lund, Sweden
christoph.reichenbach@cs.lth.se

Abstract

Datalog has emerged as a powerful tool for expressing static program analyses. Program analysis researchers have built nontrivial code bases in Datalog, but tool support for working with Datalog itself has been lacking. In this paper, we introduce MetaDL, a language extension to Datalog that enables source-level Datalog program analysis within Datalog. We describe several program analyses implemented in MetaDL and report on initial experiences. Our findings show that the language is effective for real-life Datalog analysis and can simplify working with Datalog source code.

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, Hampus Balldin, and Christoph Reichenbach. 2019. MetaDL: Analysing Datalog in Datalog. In *Proceedings of the 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3315568.3329970

1 Introduction

Declarative programming is a powerful approach for program analysis [4], and Datalog is playing a key role in this development [3], especially for points-to analysis [2].

Datalog offers concise notation and eliminates the need for manual management of *worklists* [11], a common feature in imperative program analyses. In imperative implementations, when multiple analyses are mutually supportive [7], each component analysis must interact with other analyses' worklists, which breaks modularity and complicates development and experimentation. Datalog's *semi-naïve evaluation strategy* [10] automates worklist management, freeing developers to focus on analysis logic.

Today, researchers have built Datalog-based analyses with thousands [8] or even tens of thousands [2] of rules. Understanding and working with such code bases can be challenging, as Datalog has no notion of data hiding or modularity: all information is global by design.

To help developers manage Datalog code bases, we are developing MetaDL, a program analysis tool for Datalog in Datalog. MetaDL programs can read other Datalog programs-under-analysis into queryable Datalog relations and use syntactic pattern matching to access these relations concisely. For example, we might compute a relation $\textsc{Arity}(pn, a)$, relating predicates pn to their parameter count (*arity*) a with rules like the following (see Section 3 for the full example):

$$\textsc{Arity}(pn, \$i) \text{ :- } [\ldots, \$p(\ldots, \$i : \$x), \ldots \text{ :- } \ldots], \mathrm{ID}(\$p, pn).$$

This rule matches *metavariables* ($\$p$, $\$i$, $\$x$) to code from the program under analysis, for every instance of the syntactic pattern enclosed in square brackets. Here, $\$p$ matches occurrences of Datalog head literals (predicates with arguments) in a program, $\$x$ matches the right-most argument in each matching literal, and $\$i$ is the numeric index of $\$x$ in the argument list to $\$p$. The *gaps* (…) describe sequences whose content we ignore. Outside the pattern, the literal $\mathrm{ID}(\$p, pn)$ extracts the name of the predicate bound to $\$p$ into pn. Similar queries make it easy to e.g., find all locations in which a predicate occurs on the left-hand side of a rule, identify all predicates that don't contribute to interesting output, or identify inefficient code or refactoring opportunities.

```
EDB("direct-superclass.csv", 'Superclass).
EDB("direct-interface.csv", 'ImplementsInterface).

Supertype(t, p) :- Superclass(t, p).
Supertype(t, p) :- ImplementsInterface(t, p).
Supertype(t, a) :- Supertype(t, super), Supertype(super, a).

OUTPUT('Supertype).
```

Figure 1. Datalog application code example: compute the transitive supertype relation for a Java-like language.

SOAP '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6720-2/19/06…$15.00
https://doi.org/10.1145/3315568.3329970

2 Background

MetaDL is an extension of Datalog, a declarative language that computes *relations*, effectively database tables without duplicate rows (i.e., with set semantics), from other relations. Each relation is bound to a *predicate symbol* that represents the relation in the program, and in the following we will use the two terms interchangeably.

As an example, consider Figure 1, which shows a program that computes the table of all subtypes given input data that describes a program in a Java 1.4-style language. Line 1 loads a relation from the file `direct-superclass.csv` into a relation with predicate symbol Superclass, and line 2 does the same for ImplementsInterface.

Line 4 then specifies that *any pair* ⟨t, p⟩ *that is in the relation Superclass must also be in the relation Supertype*. Such rules (or *horn clauses*) take the general form

$$P_1(\overline{x_1}) \text{ :- } P_2(\overline{x_2}), \ldots, P_k(\overline{x_k}).$$

where the P_i are predicate symbols and the $\overline{x_j}$ are sequences of variables and constants. Semantically, such rules are right-to-left implications: for all substitutions ρ from variables in $\overline{x_2} \ldots \overline{x_k}$ to constants, if we can show that the *body literals* $P_2(\rho(\overline{x_2})), \ldots, P_k(\rho(\overline{x_k}))$ are true, then the *head literal* $P_1(\rho(\overline{x_1}))$ must also be true. To show that a literal is true, we either look it up in tables loaded from disk (as in lines 1 and 2) or (recursively) derive it from any of the rules in the program. When $k = 1$, $P_1(\overline{x_1})$ is always true. In this case, we can omit the ':-' symbol, as in lines 1 and 2.

Lines 4 and 5 therefore copy all tuples ⟨t, p⟩ from Superclass and ImplementsInterface into Supertype, computing the union of these two relations. Line 6 then computes the transitive closure of the Supertype relation.

Finally, line 8 specifies that the computed Supertype relation should be written to disk once computed.

Like most Datalog systems, MetaDL adds features for arithmetic, string operations, multiple head literals, and negation.

Negation introduces a semantic complication, as it allows us to write self-contradictory rules such as $\mathrm{A}(x)$:- **NOT**($\mathrm{A}(x)$). Contradictions can also arise through longer chains of reasoning. We follow existing tools (and mathematical tradition) in requiring negation to not be recursive, i.e., whenever predicate P depends on a negated predicate Q, we must be able to fully compute Q before computing P. This process is called *stratification* (Section 4.3).

3 MetaDL

MetaDL adds a number of language features to simplify program analyses of Datalog programs. Consider the program in Figure 2, which checks that all predicates in the input program have the same arity and reports all disagreements in the relation ArityError.

Line 1 illustrates the IMPORT *pseudopredicate*, which loads an external Datalog program into a single predicate (Program). Pseudopredicates, which also include EDB and OUTPUT

```
1  IMPORT("input-program.dl", 'Program).
2
3  analyze('Program) {
4    Arity(p_name, a, loc) :-
5        [ … :- … , $p(… ,$i:$x), … . ],
6        BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
7    Arity(p_name, a, loc) :-
8        [ … :- … , NOT($p(… ,$i:$x)), … . ],
9        BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
10   Arity(p_name, a, loc) :-
11       [ … , $p(… ,$i:$x), … :- … . ],
12       BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
13   Arity(p_name, a, loc) :-
14       [ … , $p(… ,$i:$x), … . ],
15       BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
16 }
17 ArityError(p, loc_i) :- Arity(p, i, loc_i),
18     Arity(p, j, loc_j), NEQ(i, j).
19 OUTPUT('ArityError).
```

Figure 2. Program to check that the arities of predicates used within a program are consistent.

from Figure 1, follow the syntax of Datalog predicates but have special semantics (Section 3.2). As a result of Line 1, the predicate Program represents a complex relation that encodes the structure of the input program (Section 3.3).

While it is possible to analyse Datalog programs by directly accessing this *representative relation*, we provide a more concise syntax for program access in the form of syntactic patterns. From the programmers' perspective, a pattern describes the syntax that they want to match; our system rewrites this pattern into relational Datalog constraints.

For example, Line 5 uses the pattern

```
[ … :- … , $p(… ,$i:$x), … . ]
```

This pattern will match any positive (non-negated) Datalog literal that occurs on the right-hand side of a Datalog rule. Names preceded by dollar signs are metavariables. If we apply this rule to the code in Figure 1, `$p` will match the four literals occurring on the right hand side of rules, `$x` will match their right-most argument, and `$i` will match the index of the right-most argument in the parameter list (always 1 in that example, since our offsets start at 0 and all relations in Figure 1 are binary). For instance, in Line 6 of Figure 1, the above pattern would match twice, once for each Supertype literal on the right hand side, and `$x` would match the variables `super` (for Supertype`(t, super)`) and `a` (for Supertype`(super, a)`).

Returning to the analysis in Figure 2, we now increment `$i` by one and assign the result to a (BIND`(a, $i+1)`). Finally, we read the name of the predicate `$p` into `p_name`. (using ID`($p, p_name)`) and its source location into `$loc` (using SRC`($p, loc)`).

We provide three more rules that extract arity from the remaining sources of arity information: negated literals (line 7), head literals (line 10; such literals cannot be negated) and head literals in rules that omit the `:-` symbol (line 14).

We give a full overview of our pattern rules in Section 3.4.

MetaDL programs can process multiple input files at once (e.g., for code differencing or dependency tracking). We provide **`analyze`**`('`*P*`) { … }` blocks (lines 3–16), where *P* is a representative relation. Within the curly braces of such a block, the scope of all patterns and related pseudopredicates (e.g., ID) is set to exactly the program(s) represented by *P*.

3.1 Types

MetaDL has a simple static type system with type inference (Section 4.4). Each predicate *P* must have a fixed arity, and each argument must be of exactly one type. We support three types: *Int* for integers, *String* for strings, and *PredRef* for *predicate references* of the form `'P` (where P is a predicate).

We use such predicate references for input and output, but they can also communicate information between programs under analysis and MetaDL analyses (Section 4.2).

We represent the AST nodes of programs under analysis as integers (Section 3.3). Metavariables in patterns thus always bind to an integer, and developers can use pseudopredicates to extract information from these node IDs.

3.2 Pseudopredicates

MetaDL provides several pseudopredicates. EDB, IMPORT, and OUTPUT, which interface with the harddisk, require constant parameters. Several other pseudopredicates (EQ, NEQ, GT, …) test for (in)equalities. The special pseudopredicate BIND allows evaluating expressions. For example, BIND(x, y + 2 * z) will compute $y + (2 \cdot z)$ and bind the result to `x`. The first argument to BIND must be a variable.

The remaining pseudopredicates function like regular Datalog predicates, but only within **analyze** blocks:

- ID`(n, name)` relates AST nodes `n` to their `name`s, for nodes with names (i.e., predicates and variables).
- SRC`(n, loc)` relates AST nodes `n` and their source locations `loc`.
- STR`(v, s)` and INT`(v, i)` relate arguments to their constant string or integer values.
- REF`(v, n)` extracts the predicate symbols from predicate references.
- EXPR`(expr, i, subexpr)` relates expressions `expr`, which can occur in the BIND pseudopredicate, and their subexpressions `subexpr` at (zero-based) index `i`. For instance, the expression '`x + 7`' will have subexpression '`x`' at index 0 and '`7`' at index 1.

3.3 Relational Representation of Datalog Programs

Figure 3 captures the in-memory representation of the AST produced by parsing the following Datalog rule:

```
SuperTy(t, ancestor) :-
  SuperTy(t, super), SuperTy(super, ancestor).
```

When importing an AST, we assign every node in the AST a unique identifier and relate these node IDs in predicates whose names reflect the MetaDL AST (e.g. Rule, List, Atom, Variable, etc.). The relations for terminal nodes (e.g., Variable) relate node IDs to node contents (e.g., the variable name), while relations for nonterminal nodes capture the AST structure. Our encoding scheme for AST nodes uses triples $\langle p, i, c\rangle$, where p is the parent node ID, i is the (zero-based) child index, and c is the child node. With this scheme, the AST in Figure 3 is represented by the tables in Figure 4.

This representation uses a nontrivial number of relations. This in turn is at odds with our desire to provide a simple interface to the IMPORT pseudopredicate, so we currently compress all such relations into a single relation and decompress that relation again for processing **analyze** blocks.

3.4 Pattern Matching for Analysing Datalog

We currently support two forms of patterns: one for matching rules with the ':-' symbol, and one for rules without. The language accepted inside the patterns is Datalog extended with metavariables, index metavariables, and gaps.

Metavariables bind to predicate symbols, variables, constants (including predicate references), and expressions, and always start with the symbol '`$`'. Metavariables are the sole mechanism for directly connecting information from a pattern to literals outside of the pattern, where they behave identically to regular variables.

When a metavariable `$x` is part of a sequence of literals or parameters, it has an associated index that is accessible via an index metavariable, prefixed by '**:**' (e.g., `$i:$x`).

Metavariables allow limited variability in our patterns; for instance, the partial pattern `$p($v, $w)` will match any positive literal with two arguments of any kind. However, metavariables by themselves are insufficient to match literals with an unknown number of arguments , or rules with an unknown number of literals. Therefore, we also allow *gaps*.

In their simplest form (e.g., `$p(… )`), gaps specify that we permit any number of elements in an argument list or a list of literals. When gaps are adjacent to metavariables or literal Datalog code, they also relax positioning constraints.

If an element is adjacent to a gap on only one side, then the element's position is fixed relative to its neighbouring element. For example, `[ $p($v, … , $w).]` matches a single literal and binds `$v` to the first parameter and `$w` to the last parameter. If the matched literal is unary, then `$v = $w`.

If an element has gaps both to its left and to its right, its position is unconstrained in the list that it is a part of. This is a conscious design decision to allow patterns such as

```
[ … , P(… ), … , Q(… ), … . ]
```

to match predicates `P` and `Q` in any order (even if `Q` appears before `P`). If the order is significant, programmers can use index

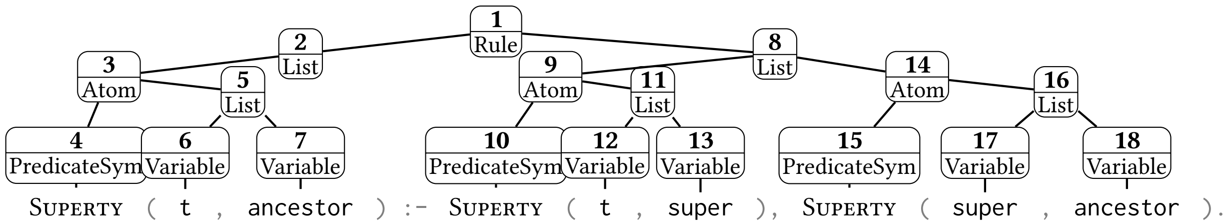

Figure 3. In-memory AST representation of a Datalog rule.

Rule		
1	0	2
1	1	8

Atom		
3	0	4
3	1	5
9	0	10
9	1	11
14	0	15
14	1	16

List		
2	0	3
5	0	6
5	1	7
8	0	9
8	0	14
11	0	12
11	1	13
16	0	17
16	1	18

PredicateSym	
10	"Superty"
15	"Superty"
4	"Superty"

Variable	
12	"t"
13	"super"
17	"super"
18	"ancestor"
6	"t"
7	"ancestor"

Figure 4. Datalog table representation of the AST in Figure 3.

metavariables (e.g., in `[ … ,$i:P(… ),… , $j:Q(… ),… .]`) to enforce the order by requiring the inequality `LT($i, $j)`.

3.4.1 Relational Representation of Patterns

We implement pattern matching by rewriting patterns into conjunctions of Datalog literals. The translation scheme is analogous to generating representative relations of imported programs (Section 3.3), with the main differences being that we (a) preserve metavariables throughout the translation and (b) introduce fresh variables for parts of the patterns that we do not wish to match (as part of gaps).

For example, recall this pattern from Figure 2:

```
[ … :- … , $p(… ,$i:$x), … . ]
```

We translate this pattern to the following conjunction:

```
1 Rule(v0, 0, v1), List(v1, 0, v4),
2 Rule(v0, 1, v2), List(v2, vj, v6), Atom(v6, 0, $p),
3 Atom(v6, 1, v3), List(v3, $i, $x),
4 BIND(v8, $i + 1 ), NOT(ListProj(v3, v8))
5 # Helper predicate:
6 ListProj(n, i) :- List(n, i, ignore).
```

All the variables (excluding metavariables) in lines 1–4 are fresh. Line 1 binds `v0` to a rule that has at least one predicate in its head. Line 2 asserts that the same rule `v0` has a child at index `vj`, and that the child must be an atom `v6` with predicate symbol `$p`. Here, `vj` is an implicit index variable. Line 3 binds `$x` to a term at position `$i` in the list of terms in atom `v6`. Line 4 ensures that that term has no right sibling (at offset `$i` + 1), as our example pattern requires `$x` to be in a rightmost position. ListProj is a helper predicate (line 6).

4 Applications

To examine the utility of our approach, we have built several program analyses in MetaDL, of which we report on five: arity checking (Figure 2), Cartesian product checking, deprecation checking, stratification, and type inference.

4.1 Checking for Cartesian Products

State-of-the-art Datalog engines like Soufflé [8] use an eager evaluation strategy. This means that rules such as

```
P(x, y) :- Q(x), R(y) .
```

can be wasteful: given j elements in Q and k elements in R, we must compute a table of $j \times k$ elements. If we instead eliminate the above rule and replace $P(x, y)$ on the right-hand side of all remaining rules by $Q(x), R(y)$, we can avoid this cost.

We have written a static checker that detects such projections, reporting any that are consistent across all left-hand side occurrences of a given predicate symbol. Our checker reports both light warnings (one projection) and serious warnings (two or more projections), using a total of 22 rules and 5 syntactic patterns. To illustrate, its final rule is:

```
CartesianProjectionWarning(p_name, $i, $j) :-
    VarProjectedN(p_name, $i, q_name),
    VarProjectedN(p_name, $j, q2_name),
    NOT(ProjectionIndicesSharedN(p_name, $i, $j)).
```

with VarProjected determining that in all rules with the predicate named `p_name` on their left hand side, index `$i` (resp. `$j`) a projection from one fixed index of right-hand-side relation `q_name` (resp. `q2_name`), further implying that the right-hand-side relation is not filtered in any way. The last line ensures that `$i` and `$j` are not only distinct but also always come from different predicates on the right-hand side; this check is slightly more fine-grained than ensuring that `q_name` and `q_name2` are distinct and will also capture e.g. `A(x, y) :- B(x), B(y).`

We have tested this checker on a self-contained miniature version of DOOP (170 rules) but found no interesting issues. We expect that such static checkers will be most useful during development of new Datalog code.

4.2 Deprecation Checking

Predicate references allow us to implement a light-weight Java-style deprecation checker:

```
Depr(p) :- [ Deprecated($p) . ], REF($p, p).
Warn(p, l) :- [ … :- … , $p(… ), … . ],
```

```
analyze('Program) {
  DirectDep(p_name, q_name) :-
     [ … ,$p(… ),…  :- … ,$q(… ), … .],
     ID($p, p_name), ID($q, q_name).

  DirectDepNeg(p_name, q_name),
  DirectDep(p_name, q_name) :-
     [ … ,$p(… ),…  :- … ,NOT($q(… )), … .],
     ID($p, p_name), ID($q, q_name).
}
Dep(p_name, q_name) :- DirectDep(p_name, q_name).
Dep(p_name, q_name) :- Dep(p_name, rn),
  DirectDep(rn, q_name).

SameStratum(p_name, q_name) :- Dep(p_name, q_name),
  Dep(q_name, p_name).
ParentStratum(p_name, q_name) :-
  DirectDep(p_name, q_name),
  NOT(SameStratum(p_name, q_name)).

Error(p_name, q_name) :- DirectDepNeg(p_name, q_name),
  SameStratum(p_name, q_name).
```

Figure 5. Stratification of Datalog in MetaDL.

```
ID($p, p), Depr(p), SRC($p, l).
```

If Deprecated('P). occurs in a program, positive references to P in the body will trigger a warning.

4.3 Stratification

Stratification is part of semi-naïve Datalog evaluation. The purpose of stratification is to (i) ensure that no relation P depends on the negation of P, or the negation of a predicate that depends on P, and (ii) construct an evaluation order over all predicates that will produce the correct result.

Stratification computes a list of *strata*, where each stratum is a set of predicate symbols that depend only on the same stratum and on all previous strata. A stratum contains at least one predicate but may contain more, if the predicates have mutual dependencies. Figure 5 gives a stratification algorithm for standard Datalog, in MetaDL.

In this figure, we first compute direct dependencies (both positive and negated) between predicates, then the transitive closure of these dependencies, Dep. We then compute which predicates must be evaluated in the same stratum due to circular dependencies, SameStratum, and the set of predicates that need to be evaluated in the parent stratum of the stratum represented by a predicate, ParentStratum. Finally, we check that no stratum has a negated dependency on itself and report violations in Error.

4.4 Type Inference

A MetaDL predicate is well-typed iff each of its arguments is used consistently with exactly one of the three MetaDL

```
1  analyze('Program) {
2    # Infer types from ground terms in facts (strings)
3    PredType(p_n, $i, "String") :-
4      [… ,$p(… ,$i:$v,… ),…  .], ID($p, p_n), STR($v, x).
5     — analogous rules omitted —
6    # Propagate types through variables
7    PredType(q_name, $j, t) :-
8      [ … :- … , $p(… ,$i:$v,… ), … ,
9        $q(… ,$j:$w, … ), … . ],
10     ID($p, p_name), ID($q, q_name),
11     ID($v, v_name), ID($w, w_name),
12     EQ(v_name, w_name), PredType(p_name, $i, t).
13    — analogous rules omitted —
14   # Compute the term indices for each predicate
15   TermIndex(p_name, $i) :-
16     [ … , $p(… ,$i:$v,… ), … .], ID($p, p_name).
17    — analogous rules omitted —
18 }
19 IsTyped(p_name, i) :- PredType(p_name, i, x).
20 IncompleteType(p_name, i) :- TermIndex(p_name, i),
21       NOT(IsTyped(p_name, i)).
22 InconsistentType(p_name, i) :- PredType(p_name, i, t1),
23       PredType(p_name, i, t2), NEQ(t1, t2).
```

Figure 6. Highlights of Datalog type inference in MetaDL.

types (*Int*, *String*, *PredRef*). A MetaDL program is well-typed iff all predicates that occur in it are well-typed.

MetaDL does not have special syntax for type declarations, but developers can set types via rules that never trigger:

```
P(0, "", 'P) :- NEQ(0, 0).
```

The above ensures that P is of type ⟨*Int*, *String*, *PredRef*⟩.

In Figure 6 we present part of an implementation of type inference. PredType(p, i, τ) defines a relation between each predicate symbol p, argument index i, and type τ that may occur at this argument index. Lines 2–4 show how we extract type information from string constants; the process is analogous for other constants and other locations in which literals occur. Lines 7–12 show how we propagate type information across body literals; the process for head literals is analogous. Finally, predicate IncompleteType(p, i) (lines 20) checks if predicate p at parameter index i lacks type information, and predicate InconsistentType (lines 22) checks if it has contradictory type information.

Type inference in rules containing the BIND and EQ pseudopredicates and arithmetic expressions can be described similarly, using the EXPR pseudopredicate.

5 Implementation

Our implementation of MetaDL is based on the Jast-Add [4] extensible compiler generator. It consists of a 'baseline' Datalog implementation and a separate MetaDL language extension module that relies on JastAdd's rewriting and nonterminal attribute features to transform `analyze` blocks and patterns to plain Datalog.

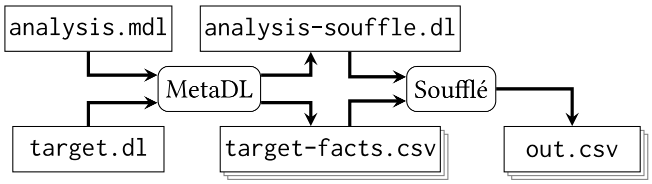

Figure 7. Evaluation strategy for MetaDL when using Soufflé as external Datalog engine.

Our system has its own Datalog backend, using the *naïve* evaluation strategy [10]. We only use this mechanism for the pseudopredicate IMPORT, then defer to an external Datalog backend (currently the Soufflé system [8]). We serialise the current rule set and all internal facts (especially our representative relations) into a backend-specific format, run the backend engine (Figure 7), then read back the results.

We have experimented with our analyses on our own code, on a self-contained miniature version of DOOP (437 lines, 170 rules) that we have ported to MetaDL, and on tests and synthetic benchmarks. For example, our checker from Section 4.1 can analyse the miniature DOOP in around two seconds; growing the target program ten-fold (1700 rules) still allows us to finish in under ten seconds. Despite being in an early stage of development, our tool is practical for analysing medium-sized code bases.

6 Related Work

Program analysis in Datalog has been an area of active research at least since Whaley and Lam [12], though their system required substantial manual representation tuning. Later systems based on LogicBlox [1, 2] and Soufflé [8] scaled more easily. While program analysis in the latter systems has focussed on backend properties, the CodeQuest system [3] demonstrated the formalism's utility for front-end analyses. Unlike ours, the above systems targeted Java or similar general-purpose languages.

The use of pattern matching has a long tradition in the functional programming community, though we are not aware of support for gaps and indices for program analysis in the same vein as our system. Coccinelle [6] supports ranges (including recursive nesting) for analysing C programs.

Analysing programs of a given language within the same language was a central topic in LISP and is also supported in other languages, primarily with the goal of supporting metaprogramming [5, 9]. While our goal is to support similar facilities in MetaDL in the future, the need for stratification raises hurdles towards offering full metacircularity.

7 Conclusions and Future Work

We have presented MetaDL, a Datalog extension for loading, analysing, and syntactic pattern-matching over Datalog programs. Our initial results show that the system can concisely express a variety of interesting program analyses and run them in a practically useful time frame. In future work, we plan to extend our pattern matching support to allow metavariables to match more syntactic constructs (including entire rules) and enable Datalog code transformation, using an extended version of our quotation syntax.

Acknowledgments

This work was partially supported by Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation. We thank the anonymous reviewers and the members of the Software Development and Environments group at Lund University for their valuable feedback.

References

[1] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[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] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. CodeQuest: Scalable Source Code Queries with Datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/11785477_2

[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] Ralf Lämmel and Simon Peyton Jones. 2005. Scrap Your Boilerplate with Class: Extensible Generic Functions. *SIGPLAN Not.* 40, 9 (Sept. 2005), 204–215. https://doi.org/10.1145/1090189.1086391

[6] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. 2010. Finding Error Handling Bugs in OpenSSL Using Coccinelle. In *European Dependable Computing Conference*. Valencia, Spain, 191–196. https://doi.org/10.1109/EDCC.2010.31

[7] Jonas Lundberg, Tobias Gutzmann, Marcus Edvinsson, and Welf Löwe. 2009. Fast and precise points-to analysis. *Information and Software Technology* 51, 10 (2009), 1428 – 1439. https://doi.org/10.1016/j.infsof.2009.04.012 Source Code Analysis and Manipulation, SCAM 2008.

[8] 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

[9] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. 2006. A Monadic Approach for Avoiding Code Duplication when Staging Memoized Functions. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '06)*. ACM, New York, NY, USA, 160–169. https://doi.org/10.1145/1111542.1111570

[10] Jeffrey D. Ullman. 1989. Bottom-up beats top-down for datalog. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM Press, New York, NY, 140–149. https://doi.org/10.1145/73721.73736

[11] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., Riverton, NJ, USA, 214–224. https://doi.org/10.1145/

1925805.1925818

[12] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog and Binary Decision Diagrams for Program Analysis. In *Proc. of the 3rd Asian Symp. on Prog. Lang. and Systems (Lecture Notes in Computer Science)*, Kwangkeun Yi (Ed.), Vol. 3780. Springer-Verlag.