Datalog Implementation in JastAdd

Hampus Balldin
C13, Lund University, Sweden
dat12hba@student.lu.se

Abstract

This paper presents the implementation of a new programming language belonging to the Datalog family. A short background on Datalog, its theoretical foundations, and its applicability for practical problems is introduced. A number of extensions to the core Datalog language are presented, the most novel of which is to allow predicates to be used as terms in atoms. As will be shown, this enables a compact description of various meta properties such as encoding what tables to output, or to give static types to predicates.

1 Introduction

Datalog is a syntactically simple declarative language that enables expression and evaluation of certain first-order logic propositions. From its inception in the nineteen-eighties, Datalog languages saw substantial interest from the academic community into the early nineteen-nineties[7]. The effort's primary drive was to create knowledge based systems in which new facts can be generated through first-order rules of inference. At the time, this had applications in both artificial intelligence and as a complement to the traditional relational database querying systems such as SQL[5][3].

After a time of cooling interest, Datalog has emerged again as an attractive way to express complex inter-dependencies[7]. A notable example is from Program Analysis where frameworks such as Doop[9] make use of Datalog to derive e.g. call-graph and points-to information, both of which typically have mutually recursive dependencies in languages using dynamic dispatch.

There are currently many Datalog implementations, including Souffle[8], IRIS[4], and BDDBDDB[14]. The implementations provide different evaluation methods and different extensions to the core Datalog language.

This paper describes a common front-end for Datalog cross-compilation which we will call $Datalog^M$. $Datalog^M$ aims to provide meta-predicates which facilitate a compact Datalog description language that may be internally evaluated or cross-compiled to another Datalog implementation. The current implementation supports cross-compilation to Souffle as well as internal interpretation. The list of supported meta-predicates is far from complete, but some progress is made through the inclusion of so called predicate references which allow predicate terms to reference other predicates (do not worry if these notions are unfamiliar, they will shortly be explained).

$Datalog^M$ is implemented using JastAdd[6]. JastAdd is a meta-compilation system that enables encoding of arbitrary graphs on top of an abstract syntax tree (AST). Information is propagated through the AST using so-called Reference Attribute Grammars[6]. JastAdd also supports aspects, which allows weaving of methods and class fields from different source locations into a single generated class. It is thus straightforward to extend the generated AST classes with additional properties. In particular, it conveniently permits the incremental addition of support for source-to-source compilation to different Datalog implementations.

This paper assumes no previous exposure to Datalog. The next subsection describes the core language and gives a brief theoretical background. Section 2 describes the core Datalog implementation. Section 3 describes some common Datalog extensions that have been implemented and also introduces meta-predicates. Finally, section 4 contains an evaluation of the implementation.

1.1 Core Language

There are many flavors of the Datalog language but they all build on a common core. A *program* P consists of a set of *Horn clauses* $H_1, \dots H_n$. A horn clause has a *head* and a *body*. The head is a single *atom* and the body is a sequence of atoms. An atom is identified by a *predicate symbol* and a sequence of *terms*. An example of a propositional rule (i.e. a rule with atoms that have no terms) is shown below:

$$A \text{ :- } B_1, B_2, \dots B_m$$

Above is a single horn clause (hereafter called a *rule*). It has head A and body $B_1, B_2, \dots B_m$. The intuitive meaning of the rule is that if the conjunction of all the atoms in the body are true then we conclude that the head A is true.

Datalog deals not only with propositional rules, but allows a restricted range of first-order propositions where each atom is associated with a sequence of terms. A term is either *variable* or *constant*. An atom that contains only constant terms is called a *ground atom*. We further partition the predicates into extensional (EDB) and intensional (IDB). The EDBs are all predicates that are taken as input from an external database. The IDBs are the predicates that are not EDB and are intensionally defined through rules. The EDBs introduce *facts*, i.e. ground atoms.

- The set of all constants in all facts is called the *domain* and is denoted Ω.

Course paper, EDAN70, Lund University, Sweden
January 16, 2019.

- The set of all facts is called the *active database instance* and is denoted I.

There are three main semantic interpretations of Datalog: model-, fix-point-, and proof-theoretic semantics [7]. The model theoretic semantics gives meaning to the clauses but does not directly yield an algorithmic approach to derive all tuples. Both fix-point- and proof-theoretic semantics yield straightforward algorithms. The implementation described in the next section is based on the fix-point approach and is outlined below. Proof-theoretic semantics is not described further.

Model-theoretic Semantics

A *model* of a Datalog program P is a consistent (satisfying all rules of P) extension of the initial EDB facts. Each rule is interpreted as a universally quantified first-order statement. For example, below is given a rule and its corresponding semantic interpretation.

$$A(x, y, "C") \text{ :- } B_1(x, "C"), B_2(x, y)$$
$$\forall c_1 \in \Omega.\ \forall c_2 \in \Omega.\ \ B_1(c_1, "C") \wedge B_2(c_1, c_2) \implies A(c_1, c_2, "C")$$

An inference algorithm attempts to find the *minimal model*, i.e. a model m of P such that for any other model m' of P, all facts of m are contained in m'. In practice this means that an inference algorithm should only add a fact if it is required by the semantics of a rule (even if adding the fact may lead to an extended model of P).

Fixpoint-theoretic Semantics

Begin with the set of all facts in the active database instance I^0. The set of new facts that can be derived (under model-theoretic semantics) using the rules of a program P and the existing facts in I^i is denoted Δ_i. We get the following inductive definition of I:

$$I^0 = \{\text{EDB Facts in } P\}$$
$$I^{i+1} = I^i \cup \Delta_i$$

It can be shown[7] that the minimal model is computed as I^n for n such that $I^n = I^{n+1}$. Since $I^i \subseteq I^{i+1}$ (monotonically increasing) and with the practical assumption of a finite and fixed domain, the fix-point algorithm is guaranteed to terminate.

2 Core Language Implementation

The current query evaluation mechanism for *Datalog*^{M} is a bottom-up naive[7] evaluation. It is based on the fixpoint-theoretic semantics that derives tuples from rules until no new tuples can be derived. The rule evaluation is performed using relational algebra (see e.g. [1]) and a thorough description is given in Appendix A.

2.1 Mutual Dependencies and Predicate Ordering

With multiple rules and potentially many mutual dependencies between the predicates, there is a need to find an order in which to apply the rules. Indeed, for mutually dependent predicates, all rules that may derive new facts for those predicates need to be iterated together. *Stratification*[7] is the process of clustering the predicates that need to be computed together into so called *strata* as well as to find an optimal order between the strata. The iterative fix-point algorithm is then run over each *stratum* following the computed order. The process is formalized below.

A predicate P_i *directly depends* on predicate P_j iff there exists a rule for which P_i is in the head and P_j is in the body. Let $Dep(P_i)$ be the set of predicates which P_i directly depends on. The dependency graph G_{DEP} has the set of predicate symbols as vertices and there is an edge from P_i to P_j iff $P_j \in DEP(P_i)$. A strongly connected component in G_{DEP} then contains the predicates which are mutually recursive. Such a connected component can be found e.g. using Tarjan's algorithm [10] and is called a stratum. By merging the vertices of G_{DEP} into such strata we get a graph G_{STRAT} with vertices being the strata of G_{DEP} and edges the collapsed multi-edges from G_{DEP}. By construction there exists a total order on G_{STRAT} with $S_1 < S_2$ iff $(S_1, S_2) \in Edge(G_{STRAT})$. The desired order is found by a reverse post-order search of G_{STRAT}.

2.2 Cross Compilation

In addition to internal evaluation, *Datalog*^{M} supports cross-compilation, or pretty-printing, to Souffle[12]. The compilation pipeline is shown in figure 1. First, a number of semantic checks are performed. For example, the semantic check ensures that all variables used in the head of a rule also occures in the body of the rule (the range restriction property[5]). In the next stage, the program is type checked (type-checking is described in more detail in the following section). As was mentioned in the introduction (and will be explained in the next section), *Datalog*^{M} supports meta-predicates. Souffle however has no such support so naturally it does not recognize the meta-semantics. To this end, a separate pre-process Datalog program is generated to evaluate all meta-predicates and subsequently output them as EDB files. Finally, the program is pretty-printed to a Souffle program $P_{Souffle}$. $P_{Souffle}$ declares the meta-predicates and loads them from the EDB files; the meta-predicates can then be used as ordinary predicates within the Souffle environment.

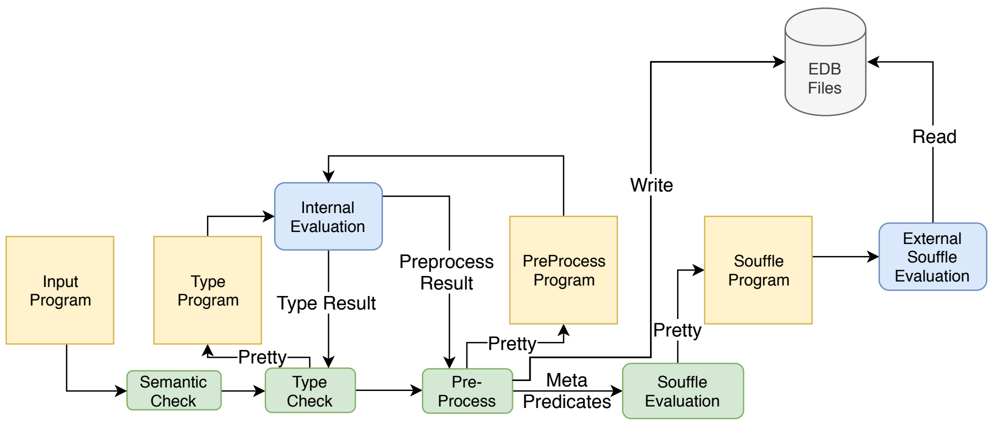

Figure 1. Souffle Printing Pipeline. **Yellow**: A Datalog Program. **Blue**: An evaluation mechanism. **Green**: A compiler stage.

3 Language Extensions

A number of language features have been added in addition to the core Datalog features. Those extensions are listed and briefly described in the following subsections.

3.1 Negation

A common Datalog extension is to allow negation ($\neg$) of atoms. However, unrestricted negation introduces semantic issues as the following example illustrates:

$$A(x) \text{ :- } \neg B(x). \qquad [r_1]$$
$$B(x) \text{ :- } \neg A(x). \qquad [r_2]$$

There are two issues with the above example. First, no unique minimal model exists: if r_1 is evaluated first then $A = \Omega, B = \emptyset$, and if r_2 is evaluated first then $A = \emptyset, B = \Omega$. Second, even if a unique minimal model exists, guaranteed termination is lost since negation removes monotonicity (adding tuples to one relation may remove tuples from another). The first issue is addressed by ensuring that all variable terms used in a negated atom are also used in a non-negated (*positive*) atom. Second, we require that each stratum retains the monotonic property, in particular this means that mutual recursive dependencies must be positive.

With the above restrictions, negation becomes a filtering rule; an occurrence of a negated atom within a rule has already been fully evaluated when that rule is considered in a stratum.

3.2 Object Creation

A special bind predicate was introduced to enable creation of new objects. Object creation extends the expressive power of the language. For example, given two numbers x and y, it is now expressible that $x + y \in \Omega$. However, the additional expressive power may lead to non-terminating programs as the following example for generating the natural numbers shows.

```
Nat(0).
Nat(y) :- Nat(x), BIND(y, x + 1).
```

3.3 Built-in Predicates and Expressions

Most (if not all) Datalog systems include various built-in predicates for discarding certain results based on some criteria. The current implementation includes the usual binary predicates $=, \neq, \leq, \ldots$. They can be used with expressions over the usual binary operators $+, -, *, /$. To continue with the Nat example, a relation that describes the first 1000 natural numbers is shown below:

```
Nat(0).
Nat(y) :- Nat(x), BIND(y, x + 1), x <= 1000.
```

3.4 Type System

The language includes a simple type system. The basic types are:

$$String : * \quad Integer : *, \quad PredRef : *$$

The types are themselves terms and the star indicates the type of a type (-term). The $PredRef$ type is used to reference predicates and forms the basis for meta-predicates (meta-predicates are introduced in greater detail in section 3.5). A term of type $PredRef$ is introduced by putting a single quote before a predicate name, e.g. $'Nat$ references the Nat relation. The type system also includes a List *type-constructor*, i.e. *List* is a function from a term of type $*$ to another term of type $*$:

$$List : * \longrightarrow *$$

The types are introduced through a special typing (meta-) predicate:

$$TYPEOF : PredRef \times List(*)$$

In this way, $TYPEOF$ relates the referenced predicate with a list of types:

$$TYPEOF('A, [t_1, \ldots, t_n]). \implies A : t_1 \times \ldots \times t_n$$

3.4.1 Type Checking and Type Inference

Type-checking and left-to-right type inference of a Datalog program P is achieved through the generation of another Datalog program P_T. For each *clause* (rule or fact) in P with head H and body B, a corresponding rule is added to P_T

with head R_i (with fresh index i) and body H, B. In addition, all facts are *lifted* to the type level and similarly all initial $TYPEOF$-facts are added as facts to P_T. Lifting of terms to type-terms is especially easy since the language already recognizes type-terms. For example, a fact [Age("Deckard Cain", 83)] in program P gets added as a corresponding type fact [Age(String, Integer)] in the type-program P_T. To support left to right type-inference, the original rules are also added without modification to P_T (with potential constants again lifted to the type-level). An example is shown in figures 2 (original program P), and 3 (transformed type program P_T). The program type-checks if and only if the solution for every relation in P_T contains exactly one tuple (i.e. has a unique type). The correctness for the type algorithm is more formally argued for in Appendix B.

Figure 2. The source program P.

```
TYPEOF('A, [String Integer]).
B(1, 2).
C(x, y, z) :- A("Deckard Cain", y), B(z, z).
D(x, y, z) :- B(x, y), A(z, y).
```

Figure 3. The transformed type program P_T.

```
A(String, Integer), B(Integer, Integer).
Typeof(PredRef, List(Type)).
C(x, y, z)     :- A(String, y), B(z, z).
D(x, y, z)     :- B(x, y), A(z, y).
Rule0(x, y, z) :- C(x, y, z), A(String, y), B(z, z).
Rule1(x, y, z) :- D(x, y, z), B(x, y), A(z, y).
```

3.5 Meta-Predicates

The language needs a way for the user to communicate certain properties about the Datalog-program P to the interpreter I. An example of such a property is what relations to load as EDBs. In turn, I makes certain information available to P which allows P to make inference on properties of itself. The information passing is realized through a collection of pre-defined atoms. The currently supported meta-predicates together with their semantics is listed in figure 6. EDB and $OUTPUT$ pass information from P to I. $ATOM$ and $PRED$ pass information from I to P. $ATOM$ contains a predicate reference for each user-defined atom. $PRED$ contains a predicate-reference for each occuring predicate. $TYPEOF$ initially provides I with information about the given types. After successful type-checks and possibly type-inference, I makes the result available to P, again through $TYPEOF$. In this way, $TYPEOF$ is a bi-directional predicate. Figures 7, 8 show two examples of possible usages for the meta-predicates. Figure 4 shows a valid usage of the $TYPEOF$ predicate. The type of B is inferred from the type of the $TYPEOF$ predicate, and the output of B is as shown in figure 5.

Figure 4. A valid program that uses the $TYPEOF$ information

```
OUTPUT('B).
B(x, y) :- TYPEOF(x, y).
```

Figure 5. Output ("B.csv") of program in figure 4.

```
'B,[PredRef List(Type)]
'OUTPUT,[PredRef]
'TYPEOF,[PredRef List(Type)]
```

Predicate	Type	Semantics
Datalog Program → Interpreter		
EDB	$PredRef \times String$	$('A, s) \in EDB$ Tuples in file s loaded into A
$OUTPUT$	$PredRef$	$('A) \in OUTPUT$ Tuples in A printed to "A.csv"
Datalog Program ← Interpreter		
$ATOM$	$PredRef$	$('A) \in ATOM$ A is a user-defined atom.
$PRED$	$PredRef$	$('A) \in PRED$ A is any occurring atom.
Datalog Program ↔ Interpreter		
$TYPEOF$	$PredRef \times List(*)$	$('A, [t_1, \ldots, t_n]) \in TYPEOF$ $A : t_1 \times \ldots \times t_n$.

Figure 6. A list of supported meta-predicates. The semantics column shows an if-and-only-if relation between the upper and lower statements.

```
OUTPUT('OUTPUT).
OUTPUT(x) :- ATOM(x).
```

Figure 7. Printout all user defined-atoms as well as the $OUTPUT$-relation.

```
EDB('EDB, "EDB.csv").
```

Figure 8. Load the tuples that describe what to load as EDB files from the external database file EDB.csv

3.5.1 Interaction with Stratification

The $OUTPUT$-predicate determines what predicates to compute and output. Thus every predicate *directly depends* on $OUTPUT$ (see section 2.1). To evaluate a $Datalog^M$ program, the relations are first partitioned into strata. The stratum containing the $OUTPUT$-predicate is evaluated first (using the fix-point algorithm) to compute the objects in the $OUTPUT$-relation. The semantics (see figure 6) forces each such object to be evaluated and subsequently printed.

By rooting the reverse post-order search in the strata corresponding to the predicate references in the $OUTPUT$-relation, the resulting order (i.e. sequence of stratum) will contain stratum S if and only if there exists some predicate reference p in the $OUTPUT$-relation such that the stratum of p

transitively depends on the S:

$$S \text{ is evaluated } \iff \exists\, p \in OUTPUT.\ Strat\ p \overset{*}{\rightarrow} S$$

The EDB-predicate too needs special attention regarding dependencies since it has the side-effect of populating predicates with objects loaded from external data base files. The following two dependency rules exist:

$$EDB('P,\ \_) \implies P \rightarrow EDB$$
$$EDB('EDB,\ \_) \implies \forall P \in PRED.\ P \rightarrow EDB$$

The first rule states that if a predicate is referenced in the EDB-relation, then that predicate directly depends on EDB. The second rule states that if the EDB-relation is self-referential, then all predicates directly depend on the EDB-relation. The current implementation does not support querying the EDB relation or updating it with a variable, i.e. the following is disallowed:

$$EDB(x,\ \_)\ \text{:-}\ A(x).$$

EDB and $OUTPUT$ may depend on each other. Consider the example in figure 9. If the file "OUTPUT.csv" contains the single entry $'A$, then the output of the program in figure 9 is a file "A.csv" containing the single entry $'A$.

```
EDB('OUTPUT, "OUTPUT.csv").
A(x) :- OUTPUT(x).
```

Figure 9. Implicit mutual dependency between EDB and OUTPUT.

4 Evaluation

The evaluation is divided into three parts: testing, performance, and expressive power. In all parts, Souffle[12] is the implementation evaluated against.

4.1 Testing

The Souffle pretty-printer (SPP) is used to output a Datalog program that may be executed by Souffle. The process of comparing Souffle output to that of the internal interpreter has been automated and a range of tests written. If the tests agree, then the SPP is said to be correct for the given test program. Assuming that the Souffle result is correct, the internal interpreter too is concluded to be correct for the given test program. The test cases are selected to cover mutual recursion, negation, meta-predicates, and the other various language extensions.

4.2 Performance

Souffle implements semi-naive evaluation[7] which is essentially the same as naive evaluation except that it utilizes the following key-insight. An instantiation of the terms of a rule may derive new tuple(s) if and only if at least one tuple that was derived in the previous iteration is used in the instantiation. Thus the number of tuples to consider can be greatly decreased.

The performance is evaluated against two examples shown in figures 10 and 11.

Figure 10. Upper bounded Natural Numbers example.

```
Nat(0).
Nat(y) :- Nat(x), BIND(y, x + 1), y <= N.
```

Figure 11. Ancestor relation example.

```
r1: Ancestor(p, c) :- Parent(p, c).
r2: Ancestor(a, c) :- Parent(p, c), Ancestor(a, p).
```

4.2.1 Theoretical Prediction

NAT Example

For the internal naive algorithm, at step k in the iteration, the Nat-relation contains k tuples. The internal implementation uses a tree-set to store the tuples and so each step takes $O(k \cdot log(k))$ time (there is no join (Appendix A) between Nat and BIND). There are a total of N steps in the algorithm, thus we get the following upper-bound for the worst-case running time:

$$\sum_{k=1}^{N} k \cdot log(k) \leq N^2 log(N) = O(N^2 log(N))$$

For semi-naive evaluation, each iteration gives a single new element to consider. The corresponding time complexity thus reduces to the order of:

$$\sum_{k=1}^{N} log(k) \leq N log(N) = O(N log(N))$$

Ancestor Example

Assume the initial parent relation:

$$Parent(P_i, P_{i+1}), i = 1 \ldots N-1$$

Then at the $k : th$ iteration of rule r_2, the $Ancestor$ relation contains $\sum_{l=1}^{k}(N-l)$ elements. The parent relation is constant with N elements. The dominating (non-indexed) *join*-operation thus has accumulated time complexity:

$$\sum_{k=1}^{N-1} \left(N \cdot \sum_{l=1}^{k} (N-l) \right) = O(N^4)$$

Similarly, the semi-naive algorithm has expected time-complexity:

$$\sum_{k=1}^{N-1} N \cdot (N-k) = O(N^3)$$

4.2.2 Experimental Results

NAT Example

The NAT example was measured for inputs in range 100 to 10000 for the internal evaluation, and in range 4096 to 536870912 for Souffle. The theoretical results was confirmed as $O(N^2 log(N))$ for the internal evaluation. For Souffle, the experimental results show $O(N)$ behavior. The $log(N)$ does not show in Souffle, most likely due to a more sophisticated relation representation[8].

Ancestor Example

The ancestor example was measured for inputs in range 5 to 260 for the internal evaluation, and in range 50 to 16384 for Souffle. The theoretical results was again confirmed as $O(N^4)$ for the internal evaluation. Souffle performed better than expected at about $O(N^{2.33})$. Again, this is most likely due to better *join*-performance which can be achieved by storing separate index data structures, thus removing the need to explicitly form the Cartesian product[8].

4.3 Expressive Power

Souffle supports all the implemented language extensions except meta-predicates and type-inference[12]. The current set of meta-predicates offered by $Datalog^M$ does not extend the expressive power in any meaningful way, but rather permits more compact and (admittedly subjectively) more beautiful descriptions of Datalog programs.

Souffle contains a number of language extensions that are not currently supported by $Datalog^M$. For example, aggregate functions (such as $COUNT$, MIN, MAX), union types (i.e. stating that a term has type A OR type B), and inbuilt functions (e.g. string operations), and more[12].

5 Related work

5.1 Souffle

Souffle implements a semi-naive evaluation[7] and performs a range of different optimizations before emitting highly templated C++ code[8]. It focuses on performance and is for example used in program analysis[9]. It does not provide meta-predicates but instead uses special directives to declare what files should be used as EDBs, what relations should be printed, and what the type of a predicate is. It does however provide a number of built-in functions and aggregates that are not supported by $Datalog^M$[12].

5.2 Other

IRIS [4][11] is an Open Source Datalog implementation written in Java. IRIS supports bottom-up evaluation strategies (both naive and semi-naive), as well as top-down (proof-theoretic semantics) strategies[11]. It was developed primarily as a research-tool for Semantic Web-related technologies and supports XML Schema data types.[13].

BDDBDDB[14] was developed at Stanford in the early 2000s and used the then novel technique to store relations as so called binary-decision diagrams (BDDs). BDDs encode relations as binary functions represented as directed acyclic graphs. The encoding allows compression of the relations and can be very memory efficient. Memory efficiency is important in e.g. program analysis, which is the main use-case that the project targeted[14].

6 Conclusion

This paper describes a working Datalog implementation with a number of extensions to the core Datalog semantics. In particular, the addition of predicate references in conjunction with a number of meta-predicates permit compact descriptions of Datalog programs. Together with cross-compilation to another more efficient Datalog implementation such as Souffle, $Datalog^M$ has the potential to grow into a useful Meta-Compilation system for specifying Datalog programs. Even so, performance is one area where $Datalog^M$ is currently lacking. The evaluation shows that two optimizations in particular would be worthwhile; implementing Semi-naive evaluation, and to add relation indexing in order to decrease the time-complexity of the *join*-operation.

Acknowledgments

Thank you Christoph Reichenbach for your many insights and helpful guidance throughout this project.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] 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

[3] Francois Bancilhon and Raghu Ramakrishnan. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. ACM, New York, NY, USA, 16–52. https://doi.org/10.1145/16894.16859

[4] Barry Bishop and Florian Fischer. [n. d.]. IRIS- Integrated Rule Inference System. ([n. d.]).

[5] S. Ceri, G. Gottlob, and L. Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.* 1, 1 (March 1989), 146–166. https://doi.org/10.1109/69.43410

[6] Torbjörn Ekman and Görel Hedin. 2007. The Jastadd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (Oct. 2007), 1–18. https://doi.org/10.1145/1297105.1297029

[7] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends databases* 5, 2 (Nov. 2013), 105–195. https://doi.org/10.1561/1900000017

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

[9] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded (Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[10] Robert Tarjan. 1972. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING* 1, 2 (1972).

[11] IRIS Team. [n. d.]. IRIS GitHub. ([n. d.]). https://github.com/NICTA/iris-reasoner last checked: 2019-01-14.

[12] Souffle Team. [n. d.]. Souffle Home. ([n. d.]). https://souffle-lang.github.io/docs/home/ last checked: 2018-12-07.

[13] W3C. [n. d.]. XML Schema Part 2: Datatypes. ([n. d.]). https://www.w3.org/TR/xmlschema-2/ last checked: 2019-01-14.

[14] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems (APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 97–118. https://doi.org/10.1007/11575467_8

7 Appendix A: Relational Algebra for Rule Evaluation

Consider the following example:

$$Order(1,2),\ Order(2,3). \qquad [r_1]$$

$$Order(x,z) \text{ :- } Order(x,y),\ Order(y,z). \qquad [r_2]$$

Rule $[r_1]$ states that the orders $1,2$ and $2,3$ holds. Rule $[r_2]$ states that the binary *Order* relation is transitive. The BUN evaluation proceeds as follows. For each rule that is evaluated, an artificial body-relation B is introduced. The body-relation will be incrementally populated and extended through the evaluation of the atoms in the body.

Initially, the relation associated with each atom is unnamed, i.e. the columns of the relation have no name-restrictions. Given a rule r, we order the atoms from the body of the rule as $A_1, \ldots, A_n$. We will consider each atom in turn and use the body relation B as an accumulator. The equations describing the rule-evaluation process is as follows (the notation is explained below):

$$\begin{aligned} B^0 &\leftarrow \top \\ B^{i+1} &\leftarrow B^i \bowtie \overline{\sigma}_{Term(A_{i+1})}\ (A_{i+1}),\ i = 0, \ldots, n-1 \\ H &\leftarrow H \cup \Pi_{Term(H)}(B^n) \end{aligned}$$

The body relation is initially assigned to $\top$ denoting an unknown relation: $\top \bowtie R = R \bowtie \top = R$. The $Term$ function gives the terms in the given atom. H is the head of rule r. A selection is then done for the atom. The special selection operator is denoted $\overline{\sigma}$. Informally it selects a set of tuples from the relation associated with the atom that satisfies the constraints imposed by the terms of the atom. As a more formal example:

$$\overline{\sigma}_{x,x,y,c} = \rho_{x/x_1, y/y_1} \circ \Pi_{x_1,y_1} \circ \sigma_{x_1=x_2, C_1=c} \circ \rho_{x_1/N_0, x_2/N_1, y_1/N_2, C_1/N_3}$$

The selection is for three variables x, x, y and a constant c. First the rename operator ρ is used to rename the columns of the relation (N_i is an artificial initial name for column i). Then the ordinary σ operator selects all tuples such that the corresponding variables and constants match under the given naming. The result of the selection is then projected (discarding duplicates and constants) using the projection operator Π, and finally the inverse renaming is performed.

The selection result is joined ($\bowtie$) with the current body relation B^i to form the next body relation B^{i+1}. In our example (assuming that r_1 has been evaluated) we get:

$$\begin{aligned} B^1 &\leftarrow \overline{\sigma}_{x,y}\ (Order) = \{(1,2),(2,3)\}_{x,y} \\ B^2 &\leftarrow B^1 \bowtie (\overline{\sigma}_{y,z}\ (Order) = \{(1,2),(2,3)\}_{y,z}) \\ &= \{(1,2),(2,3)\}_{x,y} \bowtie \{(1,2),(2,3)\}_{y,z} \\ &= \{(1,2,3)\}_{x,y,z} \end{aligned}$$

Finally we project the result of B^n and add the new tuples to the head relation:

$$Order \leftarrow Order \cup (\Pi_{x,z}(\{(1,2,3)\}_{x,y,z}) = \{(1,3)\})$$

The process is iterated until a fix-point is found, i.e. until no new tuples can be derived from the set of rules. In our example, iterating $[r_2]$ again gives no new tuples and so the *Order* relation has been computed as: $\{(1,2),(1,3),(2,3)\}$.

8 Appendix B: Correctness of Type Algorithm

For an atom A in the type algorithm, let the corresponding set of tuples be denoted A_R. The typing algorithm reports a type error in one of two cases:

$$\exists A \in PRED_{P_T}.\ \ |A_R| > 1 \qquad (1)$$
$$\exists A \in PRED_{P_T}.\ \ |A_R| = 0 \qquad (2)$$

A Datalog program P is said to be *well-typed* if:

$$\forall A \in PRED_{P_T}.\ \ |A_R| = 1 \qquad (3)$$

Let $Pos(x, A)$ give the coordinate of term x in predicate A, and $A[n]$ give type of A at coordinate n. A rule with two atoms using the same variable name gives a *type constraint*:

$$\frac{\begin{array}{c} B[Pos(x,B)] = \tau \\ \exists Rule : \ldots, A(\ldots, x, \ldots), \ldots, B(\ldots, x, \ldots), \ldots \end{array}}{A[Pos(x,A)] = \tau}\ (C\text{-}VarLeft)$$

$$\frac{\begin{array}{c} A[Pos(x,A)] = \tau \\ \exists Rule : \ldots, A(\ldots, x, \ldots), \ldots, B(\ldots, x, \ldots), \ldots \end{array}}{B[Pos(x,B)] = \tau}\ (C\text{-}VarRight)$$

Let the type of a constant term t be given by $T(t)$. We get the following constraint for constants:

$$\frac{\exists Clause\ C : \ldots A(\ldots, t, \ldots), \ldots, \quad t : Constant}{A[Pos(t,A)] = T(t)}\ (C\text{-}Constant)$$

Finally, the semantics of $TYPEOF$ is added as a rule:

$$\frac{\exists Fact :\ TYPEOF('A, \ldots, \tau, \ldots)}{A[Pos(\tau, TYPEOF)] = \tau}\ (C\text{-}TypeOf)$$

We say that a type algorithm is *correct* if the type solution satisfies the listed constraints together with the additional requirement that in a valid type assignment, each predicate is assigned exactly one type (property (3) above). *C-TypeOf* is satisfied by the type-checking algorithm by construction. We now show that it additionally satisfies *C-Constant* and *C-Var*. Together with properties (1), (2) this is enough to show correctness since (3) $\iff \neg((1) \lor (2))$.

Proposition: *The type-checking algorithm satisfies C-Constant*

A clause is either a fact or a rule. All facts are lifted to the type level and thus trivially satisfy *C-Constant.* Thus assume that have a rule with a constant use in an atom A. We will get a corresponding type-rule with head R_i with A unconditionally in the body:

$$R_i(\ldots) \text{ :- } \ldots A(\ldots, c, \ldots) \ldots$$

By the Datalog semantics, $|R_i| > 0$ if can derive that A contains a fact with the constant (lifted type) c at coordinate $Pos(c, A)$. Since there is no other clause that derives facts for R_i, this is the only way to satisfy $|R_i| = 1$ (if $|R_i| \neq 1$ then algorithm fails due to (1), (2)).

■

Proposition: *The type-checking algorithm satisfies C-Var**
Since C-VarLeft and C-VarRight are completely symmetrical it suffices to show for C-VarLeft. Assume that we have a rule R that gets transformed to a type-rule with head R_i:

$$R_i(\ldots, x, \ldots) \text{ :- } \ldots, A(\ldots, x, \ldots), \ldots, B(\ldots, x, \ldots), \ldots$$
$$B[Pos(x, B)] = \tau$$

The algorithm reports a type error iff (3) does not hold for the type-program. Thus $B[Pos(x, B)]$ has τ as the only member. By the Datalog semantics (and from the fact that no other rule populates R_i), (3) holds if and only if $A[Pos(x, A)] = \tau$.

■