Datalog for Program Analysis

Frederik Krogsdal Jacobsen

I. Introduction

Developing a program analysis is a complicated software engineering problem. One of the main issues is that finding the right trade-offs between precision and performance is difficult. To make things worse, many parts of an analysis will typically need to be modified to change the precision. Since complex analyses are quite large programs, this introduces many opportunities to make mistakes, misunderstand existing code, and complicate the implementation even further.

To alleviate some of these issues, Datalog has become more and more popular as a language to implement program analyses over the last few decades. Datalog [1] is a declarative logic programming language, which means that it is much more concise than most programming languages. Datalog implementations of program analyses are often orders of magnitude shorter than equivalent implementations in traditional programming languages such as C. While the use of Datalog was initially mostly motivated by the possibility of rapid prototyping due to its declarative nature, modern implementations of certain program analyses in Datalog are often faster, more precise, and more scalable than their traditional counterparts.

Over the last decades, many groups have experimented with and implemented various Datalog solvers and frameworks for program analysis. This review will examine the various approaches and properties of interest for Datalog-based program analysis tools, while also serving as an overview of what has been done so far, and what could be done in the future. A timeline of the projects reviewed can be found in Table I.

To the author's knowledge, no reviews of the existing tools and approaches have previously been conducted. Since interest in the topic is only increasing—with several new papers on Datalog and program conferences published at top conferences every year—the time now seems ripe to do so.

The review will begin by describing the research questions and the review protocol. Following this is a short introduction to Datalog and an example of how to implement a program analysis using it. After a short discussion of the basic approaches to evaluating Datalog programs, the rest of the review will proceed by examining tools and approaches from a number of different perspectives, followed by a conclusion and some promising avenues of further research. Appendices contain details of the search strategy, lists of venues in which future research is likely to be published, and information about where the various tools may be obtained.

II. Research questions

RQ1 What are the properties of interest for Datalog-based program analysis tools?

RQ2 Which approaches are used for state of the art Datalog-based program analysis?

RQ3 How do state of the art Datalog-based program analysis tools compare?

A. *Research question 1*

With this research question, we would like to shed light on the properties that characterize Datalog-based analysis tools. We would also like to determine the properties that distinguish the available tools and approaches.

An important property is expressivity, since some tools may extend the core Datalog features with e.g. stratified negation. An especially interesting question is whether the tools can handle any monotone framework, or if they are limited to e.g. monotone frameworks over finite powersets.

Another important property is performance. Not only is the time complexity of analysis with differing approaches interesting, but so is the question of which analyses can feasibly be implemented. Some tools may be aimed at high performance for specific categories of analyses, and we will attempt to idenfity these.

Finally, tools are only useful if they can actually be used. We will thus attempt to investigate any licensing issues related to the tools, whether the tools can actually be run on current systems, and whether the tools are in active development.

B. *Research question 2*

With this research question, we would like to determine the implementation approaches that are, or have been, commonly used for applying Datalog to program analysis. The approaches can very coarsely be divided into two categories: those involving a direct solution of the Datalog queries, and those involving a translation into some other formalism.

C. *Research question 3*

With this research question, we would like to highlight the benefits and disadvantages of the available tools and approaches, and make recommendations for which tools can or should be used for various purposes. We will do this by comparing the available tools and approaches according to the properties mentioned in the previous research questions.

III. Review protocol

The objective of the review is to assess the state of the art of Datalog-based program analysis tools, and to identify the groups working within the field. To reduce the risk of introducing bias into the review, we will explicitly specify the protocol by which the review has been conducted.

TABLE I
A TIMELINE OF PROJECTS USING DATALOG-LIKE IDEAS FOR PROGRAM ANALYSIS FROM 1998 TO 2020

Project	’98	’99	’00	’01	’02	’03	’04	’05	’06	’07	’08	’09	’10	’11	’12	’13	’14	’15	’16	’17	’18	’19	’20
SOUL	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨									
Succinct				▨	▨	▨	▨																
Jedd/PADDLE						▨	▨	▨	▨	▨	▨												
BDDBDDB								▨	▨	▨	▨												
Semmle									▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
DIMPLE										▨	▨												
DeepWeaver										▨													
BES/RWL											▨	▨	▨	▨									
LogicBlox											▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
DOOP												▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
μZ														▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
Flix																	▨	▨	▨	▨	▨	▨	▨
Soufflé																		▨	▨	▨	▨	▨	▨
Datafun																		▨	▨	▨	▨	▨	▨
IncA																			▨	▨	▨	▨	
Holmes																					▨		
Formulog																					▨	▨	▨
MetaDL																						▨	

A. Data sources and search strategy

The review will be based on journal articles and conference papers. Where practical, unpublished or informally reported results will also be included. Ongoing projects and accepted, but as of yet unpublished results will be included where possible. Only sources written in English will be included in the review.

The search strategy for identifying relevant research will consists of several steps. First, the electronic database DTU FindIt will be searched for relevant keywords such as “Datalog”, “program analysis”, and “declarative” in various combinations. To minimize publication bias, the same keywords will also be used to search the Internet using Google Search. Once a number of sources have been found in this way, the reference lists from these will be manually scanned for further relevant sources. Note that this approach will primarily yield older sources, since very few sources cite future publications. To account for this, a number of key venues for publishing within the field will be identified based on the sources found so far. The publications of these venues will then be manually searched for further relevant sources. This allows us to identify very recent publications and may also give better coverage of grey literature such as letters or commentaries, which may not be indexed by databases.

The last steps will be repeated multiple times until no further sources are found.

B. Source selection

Selection of sources will be conducted in two stages. The first stage will consist of a screening of titles and abstracts to identify potentially relevant sources. In the second stage, the selected sources will then be examined in more details to determine their relevancy.

Sources will only be assessed by the author of this study. If the results from a single experiment are reported in multiple sources, the reports will be treated as a single study.

C. Data extraction

To answer the research questions we will need to extract data from each of the selected sources. All data will be extracted by the author of this study. Due to resource constraints, authors of primary studies will not be contacted in case of missing data.

The following categories of data will be extracted from each source: identification features, approach, expressivity, performance, availability.

a) Identification features: This category includes title, author names, author affiliations, type of publication, publication venue (if applicable), and year published.

b) Approach: This category includes information about the approach used to represent and perform analyses in the source. This data will initially be relatively unstructured, since the approaches used are not known beforehand. The review protocol may be modified to facilitate a more structured data extraction approach if a number of standard approaches emerge during the review.

c) Expressivity: This category includes information about the expressivity of the formalism used in the source. First, we will investigate whether the formalism is at least as expressive as the core Datalog language. We will then investigate whether the formalism has any features extending the expressivity beyond core Datalog. We expect that a common extension will be negation (through stratification or otherwise), so this feature will be a separate data point.

A specific point of interest is whether the formalism allows analyses based on arbitrary monotone frameworks. If this is not the case, the restrictions will be noted.

d) Performance: This category includes information about both the asymptotic time complexity of analyses (if available in the source) and concrete benchmarks (if available in the source). We will focus especially on which categories of analyses can feasibly be implemented, and whether implementing those analyses require features specific to the tool at hand.

e) Availability: This category includes information about whether it is possible to actually use the tools described in each source. Since several tools are not free software, information

$$\begin{aligned}
\mathit{Program} &::= & \mathit{Fact\ Program} \mid \mathit{Rule\ Program} \mid \varepsilon \\
\mathit{Fact} &::= & P(\mathit{ConstantList}) \\
\mathit{Rule} &::= & \mathit{Atom} :- \mathit{AtomList} \\
\mathit{Atom} &::= & P(\mathit{TermList}) \\
\mathit{AtomList} &::= & \mathit{Atom} \mid \mathit{Atom}, \mathit{AtomList} \\
\mathit{Term} &::= & C \mid V \\
\mathit{TermList} &::= & \mathit{Term} \mid \mathit{Term}, \mathit{TermList} \\
\mathit{ConstantList} &::= & C \mid C, \mathit{ConstantList}
\end{aligned}$$

Fig. 1. A syntax for Datalog. P is a countable set of predicate symbols, C is a countable set of constants, and V is a countable set of variables. As a syntactic restriction, any free variables in the head of a rule *must* appear in the body of the rule.

about the restrictiveness of the available licensing schemes is interesting. We will also attempt to determine the period in which the tool has been actively maintained, since we expect that tools for which development has ceased may no longer be possible to run.

D. Data synthesis

Since we anticipate that it will be very difficult to compare any performance studies in the individual sources, a quantitative meta-analysis will not be performed. Instead, the results gathered from the sources will be interpreted through a narrative synthesis. The narrative synthesis will explore the relationships within and between the different sources to form a final set of conclusions and recommendations. This exploration will be based upon the tables of data extracted from the sources. We expect that the relationships between sources will take the form of various groupings of the approaches used and the results obtained for various analyses. We will also attempt to extract the main themes and concepts recurring across multiple sources, as well as the evolution of ideas within the field.

IV. What is Datalog?

Datalog is a declarative logic programming language. It is common to view Datalog as a purely declarative subset of the logic programming language Prolog, i.e. without function symbols and negation. Another view is that Datalog is a database extended with first-order Horn clauses—a "deductive database". A syntax of Datalog is presented in Figure 1.

A Datalog program is a sequence of facts and rules which may be used to deduce new information. The only operator is $:-$, which may be read as "if". The atom to the left of the operator is called the head of the rule, while the list of atoms to the right is called the body of the rule. A Datalog program may be queried to determine if a piece of information can be deduced from the facts and rules of the program. A query has the same syntax as a fact, but allows variables. Evauating a query the simply comes down to attempting to find an instantiation of the variables which can be deduced from the facts and rules of the program. It is common to couple a Datalog program with a database of ground facts derived from some source. The Datalog program is then called the *intensional database*, while the set of ground facts is called the *extensional database*. Note that only predicate symbols from the intensional database are allowed to occur in the head of a rule, since the extensional database is viewed immutable. The task of evaluating a Datalog program is then to compute the contents of the intensional database.

The semantics of Datalog programs can be approached in several ways. A logical semantics can be obtained by noting that each Datalog fact corresponds to an atomic formula of first-order logic. The semantics of a Datalog program can then be described as a mapping from the Herbrand base of the extensional database to the Herbrand base of the intensional database.

Another approach is to describe the semantics of Datalog using model theory. Datalog programs correspond to Horn clauses, and this may be used to define a notion of models of Datalog queries.

Finally, the semantics of a Datalog program may be defined as the least fixed point of a "inference" function, which computes all of the new information that may be deduced in one rule application using the currently known facts. The syntax of Datalog ensures that the set of facts that can be derived is finite, so the fixpoint always exists.

The syntactic restriction that all variables in the head of a rule must also occur in the body of the rule is much more limiting than it may seem at first glance. Arithmetic, for example, is not possible, since it is not possible to obtain the new symbol "2" from the symbols in "plus(1,1)". Such relations may be introduced as "built-in" predicates which are evaluated outside of the normal Datalog semantics, but care must be taken to keep the output of the Datalog program finite. It is also possible to extend Datalog with various notions of "objects" more complex than constants.

Another inconvenient restriction is that Datalog does not support negation. If we assume that all facts that do not logically follow from a set of Datalog facts and rules are false (the closed world assumption), negation may be introduced. For this to be consistent, the Datalog program must be stratified such that it is always possible to completely evaluate all predicates which occur negatively in the body of a rule before evaluating the head of the rule.

For more information on Datalog, the reader may refer to the classic paper by Ceri, Gottlob and Tanca [1].

V. An example analysis

This document is about applications of Datalog to program analysis, so let us now consider an example of how this may be done. Consider the following fragment of a program in a Java-like language:

```
A o1  = new A();
B o2  = new B();
B o3  = o2;
o2.f  = o1;
C o4  = o3.f;
```

One might be interested to know which objects the local variable o4 can point to (a "points-to" analysis). In this case, it can only point to the object pointed to by variable o1 (which we will call object a), since: variable o1 points to object a; variable o2 points to some other object (object b); variable o3 points to the same object as variable o2, which is object b; object a is written to the field f of object b; variable o4 points to object a since it is read from field f of object b.

This is a quite complicated reasoning for such a simple question, and in real programs it quickly becomes impractical to do such analysis manually. Consider now the following set of Datalog rules:

```
VPointsTo(v, h) :- New(v, h).
VPointsTo(v1, h2) :- Assign(v1, v2),
                     VPointsTo(v2, h2).
VPointsTo(v1,h2) :- Load(v1, v2, f),
                    VPointsTo(v2, h1),
                    HPointsTo(h1, f, h2).
HPointsTo(h1,f,h2) :- Store(v1,f,v2),
                      VPointsTo(v1,h1),
                      VPointsTo(v2,h2).
```

These rules encode the reasoning used above, and are enough to define the entire points-to analysis. The program itself may be encoded as the following extensional database:

```
New(o1, a).
New(o2, b).
Assign(o3, o2).
Store(o2, f, o1).
Load(o4, o3, f).
```

These facts are easy to extract from the program. The question may now be posed as the following Datalog query:

```
VPointsTo(o4, X).
```

Evaluating this query will result in an intensional database containing the fact VPointsTo(o4, a).

While this very simple analysis can be implemented in only a few lines of Datalog, implementing it in a language like C would take hundreds or thousands of lines of code. The crispness and simplicity of Datalog makes it easy to understand what the analysis does, and makes errors in the analysis easier to detect.

For an introduction to pointer analysis using Datalog, see e.g. [2].

VI. Evaluating Datalog queries

In broad terms, Datalog queries can be evaluated either bottom-up or top-down. A naïve bottom-up algorithm can be extracted directly from the fixed point semantics of Datalog by simply applying the "inference" function to every relation with every fact until a fixed point is reached. Unfortunately, this naïve algorithm has terrible performance because it typically computes large amounts of irrelevant facts. To gain some performance, a semi-naïve algorithm can be used instead. The idea is to the inference step only to those rules that involve the newly generated facts from the previous inference step, since every other fact has already been tried. This optimization can improve performance significantly, and is quite general. Variants of the semi-naïve algorithm can also be used to incrementalize queries to avoid recomputing unchanged facts when the extensional database changes.

A naïve top-down algorithm is to perform "backward chaining" in the style of Prolog. The problem is that this algorithm will compute all possible ways to deduce every fact, while we are typically only really interested in whether there *is* a way to deduce each fact or not.

A classic optimization that massively improves the bottom-up algorithm is the "magic sets" transformation [3]. The idea is to use auxiliary relations in order to ensure that rules are only when the fact that they derive is relevant for the query. The auxiliary relations explicitly encode which rules depend on each other using "magic facts" that "tag" each rule. This essentially simulates a top-down evaluation, propagating the "relevancy" through the magic sets. An additional predicate is then added to each original rule such that it can only infer new facts if another rule depends on it.

The top-down algorithm can be improved by memoize the derived results to avoid computing several ways of deducing the same fact. This optimization is usually called tabling. Variant tabling is the most common type of tabling, and reuses the answers to variants of previously encountered facts module variable renaming. A promising alternative strategy is subsumptive tabling, which reuses more answers by reusing answers that subsume a new subquery, i.e. answers for which some variable substitution deduces the query. There is evidence that subsumptive tabling may be even faster than the magic sets optimization both in theory and practice [4].

VII. The first program analyses in Datalog

Perhaps the first to use "Datalog" for program analysis was Aßmann, who introduced his edge addition rewrite system for program analysis called EARS in 1994 [5]. Aßmann affords Datalog only a brief mention, noting that the EARS system is in fact equivalent to Datalog.

Slightly later, Reps investigated how deductive databases and logic programming could be used for interprocedural program analysis [6]. His implementation used a Prolog solver to compute results, but found that this was much slower than an implementation in C.

Unfortunately, the next years did not offer much work on using Datalog for program analysis.

VIII. Datalog for program queries

Datalog is useful not only for program analysis, but also for the closely related, but simpler field of querying programs. Software developers will often search programs to understand and transform them. Traditionally, this is accomplished using plain text matching or regular expressions, but for more complicated queries, these solutions may not be enough. Additionally, automated tools may need more precise information than these tools can provide. If a query becomes sufficiently sophisticated, it may be valid to call it a program analysis instead. It is difficult to identify a clear bar separating program queries from program analyses, but the design of tools for the two vary due to the difference in intended use.

The Smalltalk Open Unification Language (SOUL for short) by Wuyts and Ducasse [7] [8] was one of the first projects to use logic programming for program querying. SOUL is an extension of Prolog which allows reasoning over programs written in Smalltalk. While the tool was also capable of program analysis, the main idea was to enforce programming conventions and design decisions automatically on large codebases. The use of logic programming allows architects to describe conventions and design patterns concisely. Another potential application was an advanced search/replace operation for semi-automated program transformations. A similar approach using SOUL itself as a case study presented in the thesis of Kim Mens, which also contains a good introduction to the SOUL system [9].

CodeQuest by Hajiyev, Varbaere, de Moor, and Volder [10] [11] [12] was another system for querying source code. CodeQuest uses Datalog as a query language for a database of source code. The implementation supports incremental updates of the database when a compilation unit is changed, enabling responsive queries at a large scale. This work was continued with the .QL (later called QL) language of the Semmle company [13] [14], but at this point the work is more focused on actual program analyses for security.

The DeepWeaver-1 tool by Falconer et al. [15] extended the idea of declarative code querying by also supporting automatic code transformations based on the results of queries. The tool thus implemented the ideas of advanced search/replace put forth by Wuyts in [7]. de Roover et al. implemented a plugin for the Eclipse integrated development environment based on SOUL to perform similar operations a few years later [16].

IX. Which analyses are Datalog useful for?

Most of the work on using Datalog for program analysis focuses on points-to analyses for object-oriented programs. These analyses are a natural fit for Datalog because they are essentially the reaching definitions problem, and thus only involve simple relations which can easily be encoded as Datalog rules. A major benefit of using Datalog is that the mutual recursion between rules is handled automatically.

For other analyses, Datalog has some important limitations. Datalog is limited to rules on relations (i.e. powersets of tuples), but many analyses operate on lattices. While any finite lattice can theoretically be encoded as a powerset by enumeration of the elements, the computational cost of doing so can be prohibitively high even for simple lattices. Madsen, Yee, and Lhoták claim that more interesting lattices cannot be encoded at all [17].

A related limitation is that Datalog has no concept of functions. Functions thus have to be encoded as relations by tabulation. This means that even if a lattice could be encoded efficiently, abstract operations on the lattice would also be inefficient. For these reasons, Datalog has not typically been used for e.g. constant propagation or interval analysis.

What Datalog really seems to excel at, then, are analyses focusing on the *structure*, and not the computational content, of programs. As examples of these, we can mention must-alias analysis [18], taint analysis [19], use-analysis of libraries [20], analyses of dynamic proxies in Java [21], and architectural conformance analyses [9] [22]. Datalog can even be used to analyze the structure of complex enterprise applications using dynamic techniques such as dependency injection and object caching [23]. One advantage of program analyses implemented in Datalog is that they are simple to compose, since the rules in each analysis can easily refer to each other. It is even possible to evaluate Datalog analyses on partial programs, then compose the results before applying further analysis [24].

Almost all of the analysis tools surveyed in this document extend Datalog in some way. Almost all of the tools support stratified negation or a variant of the idea.

Flix [17] and IncA [25] extend Datalog with support for defining lattices and operations on them, which allows for several common analyses such as constant propagation. Supporting the integer domain also makes more precise points-to analyses such as resolution of overloaded methods possible [26].

The Holmes tool by Maurer allows co-dependent analyses of compiled binaries by extending Datalog with the ability to reason about e.g. ELF files through external calls [27].

The LogicBlox engine [28] supports a wide array of extensions such as support for constraint handling rules [29], dynamic updates of the extensional database, user-defined functions [30], probabilistic and statistical programming [31], and a number of extra-logical computational features [32].

The Formulog language extends Datalog with the ability to represent and reason about SMT formulas to allow integration with SMT-based analyses [33].

The MetaDL system extends Datalog with the ability to reason about Datalog programs [34] [35]. This makes it possible to perform analyses on the analysis tools themselves to gain confidence in their correctness.

A. *Context-sensitive analyses*

When performing analyses on object-oriented languages, one of the main factors affecting the precision of the analysis is how the contexts of method calls are handled. If contexts are competely ignored—a context-insensitive analysis—all information about calls to some method will be intertwined, leading to very poor precision.

Context-sensitive analyses consider the context of each method call, but there are several different levels of precision at which this can be done. A popular choice of context is the allocation site of the object receiving the method call, resulting in what is called an object-sensitive analysis. Smaragdakis, Bravenboer and Lhoták explore the design space of object-sensitive analyses in [36]. They find that previous authors have made poor choices resulting in low precision and performance, and define a framework that captures the various possible approaches. They suggest that approximating object-sensitivity using the type of the object can collapse the combinatorial space significantly while preserving almost all of the precision of an object-sensitive analysis with the same number of context elements. They call this approximation a type-sensitive analysis. This allows the analysis to keep track of more context elements, leading to more precision and performance.

Thiessen and Lhoták have introduced an efficient representation of context elements by representing the analysis as an algebraic structure of context transformations [37]. The authors suggest that this results in a more efficient algorithm for computing e.g. type-sensitive analyses.

Performance can also be gained by using a varying amount of context elements to obtain good precision for the methods that need it, while preserving good performance for methods that need less precision. Jeong et al. have developed a data-driven heuristic to decide which methods should be analyzed with context-sensitivity and how much should be used [38]. They have also developed a greedy algorithm which can efficiently learn the parameters of the heuristic rules. Initial benchmarks show significant performance improvements.

The same authors have also demonstrated that significant precision and performance gains can be obtained by determining which context elements are "important", and maintaining only those elements [39]. They call the approach context tunneling, and use a data-driven approach to search for good heuristics to determine which context elements are important.

X. Datalog and binary decision diagrams

One of the earliest ideas for optimizing the evaluation of points-to analyses was to represent the relations implicitly using binary decision diagrams (BDDs). Jedd is an extension of the Java language that supports programming with BDDs [40] [41]. Lhoták and Hendren developed the extension and used it to implement the PADDLE framework for context-sensitive points-to analyses [42] [43] [44]. BDDs are a compact way to represent large sets. The idea is that any regularity in the sets can be exploited to compact the set, enabling scalability. The rules are encoded as operations on entire relations in BDDs. Variables in the relations are mapped into domains (groups of bit positions) in the BDDs. The ordering of these variables has a major impact on the performance of the analysis. Both the ordering of individual bits inside each domain and the interleaving of variables of different domains are factors. For points-to analyses, Lhoták and Hendren conclude that domains representing variables should come before domains representing allocation sites, and that variable and allocation site domains should not be interleaved.

While Jedd has nothing to do with Datalog per se, the same ideas were used for the BDDBDDB tool by Whaley and Lam [45] [46] [47]. BDDBDDB translates analyses expresses as Datalog programs into BDD representations. The reasoning for this is that Datalog programs are much easier to write and understand than equivalent BDDs. The translation from Datalog to BDDs is simple, but the tool automatically optimizes the programs by applying semi-naïve evaluation and attempting to find a good variable ordering for the BDDs.

The main novelty from the user perspective is that BDDBDDB was the first tool to support context-sensitive points-to analysis in a scalable way. The idea was to create a clone of each method for each context of interest, and then run a context-insensitive analysis on each clone. This would normally result in a completely intractable problem due to the number of contexts in a realistic program. Whaley and Lam propose that the contexts are so similar that BDDs can be used to compactly represent them in an efficient manner by numbering the contexts such that moving between them in the BDD representation can be done by simply adding or subtracting constants. This idea allowed the BDDBDDB tool to evaluate context-sensitive analyses on very large Java programs, something which had not previously been possible.

XI. Are implicit representation really better?

While Whaley and Lam were able to demonstrate good performance for context-sensitive analyses using BDDs to implicitly represent relations, other authors have questioned whether program analyses are actually regular enough to benefit from implicit representations [48]. Bravenboer and Smaragdakis implemented the DOOP framework for context-sensitive points-to analysis [49] [50] based on this thesis. The main objection of the DOOP authors is that implicit representations are only beneficial for analyses that are badly specified and thus highly redundant. The DOOP authors target the redundancy directly by improving the precision of the analysis. An important optimization is to discover call graphs on the fly, which enables the improved precision. The authors note that the BDDBDDB tool cannot do this due to the cloning-based approach.

The key novel optimization technique is to avoid traversal of non-deltas by ensuring that the relation deltas produced by semi-naïve evaluation bind all of the variables needed to index into other relations. Using this technique may require introduction of intermediate relations which serve to hold the result of intermediate joins.

Another optimization advocated by the DOOP authors is to intertwine points-to and exception analysis into a joint analysis [51]. Conventional exception analysis as used in e.g. PADDLE encodes throw operations as assignments to a single global variable, but this is very imprecise. The DOOP authors propose that this lack of precision leads to poor performance due to redundant analysis. They propose an exception analysis based on which objects may be thrown, which leads to a natural coupling with the points-to analysis. The authors argue that doing precise exception analysis can eliminate so much redundancy that the time needed to compute the exception analysis is well spent.

The combination of these optimizations improves performance by several orders of magnitude. Benchmarks show that DOOP is much faster than PADDLE, even for more precise analyses modeling the exception handling semantics of Java closely.

DOOP originally used the LogicBlox engine to actually compute the analyses, but was later ported to the Soufflé engine, which was the only engine to support parallelism at the time [52]. That this was possible showcases another benefit of Datalog, this being that the declarative nature of the language allows users to ignore most implementation details.

XII. Datalog and relational logic

Since Datalog can be viewed as a deductive database, one might wonder if there is a useful relationship between Datalog and relational logic. In fact, Datalog can be translated into various relational logics, e.g. SQL.

BDDBDDB was one of the first tools to take advantage of this [45]. When translating Datalog into BDDs, it first translates the Datalog rules into operations in relational algebra. These operations are then translated into BDD operations. Datalog rules can be translated into a series of join, project and rename operations from relational algebra.

The CodeQuest system uses Datalog as a query language by translating the Datalog queries into SQL operations, which are then executed using a usual relational database system [10] [11]. Recursion in the Datalog rules can be handled either through built-in recursive queries in the relational database system, or by a custom system of stored SQL procedures. The authors of CodeQuest found that the approach using stored procedures is faster for large programs.

Semmle goes one step further, and uses a custom relational database system to execute the translated Datalog queries [14]. The custom database system is optimized for nested joins and recursion, since this is what is needed for evaluation of Datalog queries. It also gains performance by not supporting database updates and transactions.

The μZ engine by Hoder, Bjørner and de Moura [53] [54] compiles Datalog rules into relational algebra operations for an abstract machine. The compiled operations also contain control and data-flow instructions which apply rules until a fixed point is reached. The compiled operations are executed by a register machine interpreter which stores relation objects in its registers. The representation of the relations themselves are left to implementations through an API. The default implementation uses hash tables, but it is also possible to use a BDD-based implementation. μZ supports user-defined abstract relations, which can be used by providing the tool with an implementation of the join, project, union, select and rename operations for the relation. The authors have found that the hash table implementation has better performance than the BDD implementation.

The Soufflé framework by Scholz et al. from Oracle Labs originally translated Datalog programs into SQL queries, which were then executed on a relational database [55]. According to the authors, this ensured more effective use of memory and disks than existing Datalog solvers. The translation uses a semi-naïve evaluation approach by keeping tables of deltas and newly generated tuples.

Like the CodeQuest/Semmle project, the authors of Soufflé also switched to a custom relational algebra [56] [57]. For this, Datalog is translated into relational operators for an abstract machine. Instead of interpreting the operations like the μZ implementation, however, the Soufflé compiler then translates these operations into a C++ program, which is finally compiled to a binary. The synthesized programs use OpenMP for parallelism, and the datastructures are tailored for every relation and every index. For relations with only a few attributes, geometrically encoded tries are used, while B-trees are used for relations with a medium amount of attributes. Relations with many attributes are stored in a blocked list with pointers to B-trees. All of these datastructures are modified to allow optimistic locking. Later, the authors created a novel lock-free trie structure for storing large, dense relations [58] to improve performance even more. The idea is to optimize specifically for the relations encountered in program analyses. Subotić et al. have implemented an automatic algorithm for selecting indices, which seems to obtain similar performance as manually optimized indexing schemes [59].

Benchmarks performed by the authors show very good performance compared to BDDBDDB, μZ, and the previous SQL implementation of Soufflé. The tool can also scale context-sensitive points-to analyses to very large programs which cannot be analyzed in reasonable time by the mentioned competing tools.

XIII. Datalog and logical equations

Another approach to evaluating Datalog queries is to translate the Datalog program into logical equations. Alpuente et al. have implemented an analysis framework called DATALOG_SOLVE, which translates Datalog programs into boolean equation systems (BESs) [60] [61] [62] [48]. The idea is to dynamically translate each Datalog rule into a BES whose local resolution corresponds to demand-driven evaluation of the analysis. The evaluation thus takes a top-down approach guided by the query, but the authors have also experimented with bottom-up evaluation [63]. The translation from Datalog to a BES is very simple: the clauses of a rule are translated into a disjunction, conjunctions in Datalog are translated into conjunctions, and variables are translated into boolean variables. The tool supports stratified negation and totally ordered finite domains. Using a demand-driven approach means that the tool only has to compute the information actually needed to answer the query, which can save time in some cases. The tool is not very optimized, but it was able to perform context-insensitive points-to analysis on real Java programs.

Bembenek and Chong have implemented Formulog [64], which extends Datalog with the ability to represent and reason about logical formulas. This makes it possible to implement symbolic execution and abstract model checking, which is not possible with pure Datalog. Formulog allows Datalog programs to reason about logical formulas through built-in functions that invoke an external SMT solver. A type system is imposed to ensure that these functions are used safely. The extension still allows optimizations such as parallelization, memoization and the magic sets optimization, and the tool is sometimes much faster than reference symbolic execution tools [33]. The authors note that their implementation is also able to gain performance using incremental SMT solvers even though the declarative nature of the language does not guarantee an evaluation order explicitly optimizing for this [65].

XIV. Parallel evaluation of Datalog

Alpuente et al. speculated early on that the evaluation of Datalog programs as resolution of boolean equation systems could be distributed over a network of workers to improve performance since the BES encoding has a regular structure and allows a topological ordering of the resolution steps [61]. They later considered the possibility of distributing Datalog using a Map-Reduce algorithm [66] [48], but never implemented either of the two ideas.

The LogicBlox engine was thus perhaps the first practical tool to support parallel evaluation of Datalog program queries [31]. The LogicBlox engine uses a novel concurrency control scheme and immutable structures to avoid locking and the need for cache coherence protocols.

The Flix language supports process-based concurrency, but does not specify how this is done [67].

The Soufflé framework uses OpenMP for parallelism [56]. The implementation synthesizes C++ programs which use the parallelization primitives provided by OpenMP to mark loops to be processed in parallel. The user is thus able to control the degree of parallelism. The tool initially used datastructures with optimistic locking to support parallel evaluation of queries [56]. The authors found that more fine-grained locking schemes required too much communication between nodes to improve performance. Later, a novel specialized datastructure combining the advantages of tries and B-trees is used to implement lock-free parallelism [58]. The authors call this datastructure a "Brie", and benchmarks performed by them show that it provides large performance gains for densely populated relations.

The interpreter for the Formulog language evaluates programs in parallel using a work-stealing thread pool [64]. Each worker thread works on one rule at a time, fully evaluating the rule against a database of currently derived facts. When a new fact is derived, the relevant rules are submitted to the pool as new work items. The authors consider following the strategy of Soufflé for further optimizations [33].

XV. Datalog and functional programming

Datalog and functional programming can be associated in several ways. First, functional languages can of course be used to implement solvers. Perhaps the first practical tool to do so is the Succinct solver suite by Nielson, Seidl and Nielson [68] [69] [70]. The solvers in the suite are implemented in Standard ML, and make full use of tail recursion and continuations to gain good performance without needing worklist algorithms or to identify strongly connected components. The logic of the Succinct solvers is the alternation-free fragment of least fixed point logic, which is more expressive than Datalog because it allows universal quantification and stratified negation. The use of a functional language also allows the solvers to live up to their name, since very little code is needed. The Succinct solvers showed that good performance can be obtained while keeping the solver as simple and easy to understand as the analyses it solves.

Alpuente et al. used the Maude language for rewriting logic to implement the Datalaude solver for Datalog in the [71] [72] [48]. The motivation is to handle metaprogramming features such as reflection in a simpler way than possible in conventional analysis frameworks. The Datalog program is translated into a functional program in the Maude language by representing rules as sets of constraints that represent every possible solution. The Maude language computes solutions using term rewriting, which corresponds to top-down evaluation of the Datalog program.

Second, Datalog can be extended with constructs from functional programming. The Flix language by Madsen, Yee and Lhoták extends Datalog with the ability to define complete lattices and monotone functions implemented in a pure functional language [17] [73] [74] [67]. Supporting general lattices as part of the language avoids the need to encode lattices as powersets, which can be very expensive even for low precision analyses. Supporting functions as part of the language makes it possible to express function with infinite domains or codomains, and also avoids the inelegant encoding of functions as tabulated relations. The user must verify that the lattices they define are actually complete lattices, but the authors have implemented an automatic verifier that can cover many cases [75]. The Flix language is implemented on the Java Virtual Machine (JVM), and allows users to access JVM types and methods when defining lattices and functions. Flix also allows other programs to access solutions through an API on the JVM, which avoids slow and brittle serialization to files.

The Datafun language by Arntzenius and Krishnaswami extends Datalog with support for higher-order functional programming [76] [77]. The idea is to track the monotonicity of transfer functions using types. The semantics limits types to be finite and distinguishes between ordinary and monotone variables to ensure that well-typed Datafun programs terminate. Recursion is only allowed for finite types or by explicitly bounding the recursion. The authors posit that this makes it easier to write fixed point computations that actually compute with the data they involve. An advantage of Datafun over Flix is that monotonicity is ensured by the type system, while Flix programmers must either hope that the verifier can handle their functions or prove monotonicity themselves. On the other hand, the higher order nature of Datafun makes "usual" Datalog optimizations more difficult to apply. The authors have defined a program transformation that implements semi-naïve evaluation for Datafun [78], but the Datafun implementation is still not very fast.

IncA by Szabó et al. is a language for defining and efficiently incrementally evaluate program analyses [79] [80]. The base language is equivalent to Datalog with stratified negation, but the authors have extended the language with support for defining lattices [81]. The authors have developed a novel algorithm supporting efficient incremental aggregation of lattice values in Datalog [25]. This algorithm is integrated into the IncA implementation by compiling the IncA program into a variant of Datalog, which is then solved incrementally using the new algorithm. Like with Flix, users need to verify that their lattices are correct, and the authors believe that the verifier from Flix could be modified to work with IncA since the expressivities of the languages are similar. Incrementality is useful for rerunning analyses on programs under development, and the authors demonstrate that their algorithm can recompute analyses even on large programs.

The Formulog language also allows definitions in a fragment of ML, but this is just syntactic sugar for Datalog rules [33] and so does not affect the expressivity of the language.

XVI. Completely declarative analyses

Most of the reviewed analysis tools only implement the analysis itself in Datalog, requiring preprocessing of the programs to create an extensional database that the analysis can actually understand.

Benton and Fischer's analysis framework DIMPLE was the first to use logic programming for every step of the analysis [82] [83]. DIMPLE operates directly on a representation of Java bytecode, allowing users to implement and modify every step of the analysis declaratively. DIMPLE is implemented in Prolog, however, and has several extra-logical features. The authors argue that implementing entire analyses declaratively allows easier and faster experimentation and development of new analyses.

The first analysis framework to describe entire analyses in Datalog was DOOP [49]. The authors argue that implementing entire analyses in Datalog is not only easier, but also improves precision and modularity.

Johnston has performed a preliminary case study on translating an existing analysis into a fully declarative analysis using LogicBlox [84].

XVII. Integrating Datalog with other tools

Code queries based on logical programming are not commonly used today. de Roover et al. [16] propose two possible reasons why:

- Specifying program queries is convoluted, and users are not familiar with logic programming
- Using the results of queries is cumbersome because users need to convert results back to program constructs

Their solution to the first problem is to allow example-driven specifications, and they propose solving the second problem by integrating tools into integrated development environments. A significant part of the activities of Semmle are based on what they call variant analysis, which is a type of example-driven specification [85]. The idea is to identify single issues in a codebase with a query, then take advantage of the high-level nature of logical programming to quickly generalize the original query into an analysis for a class of similar issues. Code query languages could also be used to enable automatic refactoring of code, following the ideas implemented in the DeepWeaver-1 tool [15].

The IncA [79] [25] [80] language, the LogicBlox engine [31], and the Datafun language [78] support efficient incremental evaluation of analyses. In many cases, this allows very fast recomputation of analyses when code is changed. This could be very useful to provide real time warnings or suggestions by integrating analyses into IDEs. This would allow software development to become more agile while also ensuring adherence to standards encoded as program analyses. The demand-driven evaluation strategy of the DATALOG_SOLVE tool [62] could enable a similar integration. Vilter has implemented a prototype IDE which allows the use of Datalog to define program transformations, visualizations and analyses [86].

The integration of the Flix language with the JVM platform allows the results of analyses to be consumed by other tools written in JVM languages [17], and this type of integration could make Datalog-based solvers easier to integrate into larger tool suites and IDEs. The Formulog language allows Datalog analyses to integrate with SMT-based analyses [33].

Zhao et al. [87] identify three limitations preventing further adoption of current declarative program analyses:

- Composition of analyses from different tools, which is difficult due to differences in representations
- Lack of information from other sources, which makes guided program optimizations difficult to implement
- The need for preprocessing, which is often coupled with some specific compiler framework and analysis

The authors believe that these problems can be solved by incorporating ontology into declarative analysis tools. They identify the lack of a systematic framework for organizing and representing various kinds of knowledge, including relations, rules, domains, hardware configurations, and so on. The authors have implemented a prototype framework called PATO to validate the idea, but have not exploited the claimed potential much.

XVIII. Strengths and weaknesses

The review covers publications and tools from the inception of the field until today. The search strategy used should prevent the review from missing any important developments, since they would most likely have been referenced by one of the covered publications. The search strategy should also prevent the review from missing any major unpublished developments.

Unfortunately, most of the tools extend Datalog in different ways, making comparisons between them difficult. Some tools are much more expressive than others, which makes fair comparison of performance difficult. It is also unclear how various extensions affect the expressivity, and whether some extensions are equivalent or perhaps form a hierarchy.

Many of the publications include performance benchmarks, but they are rarely very thorough. Additionally, the large span of time between the first and current tools means that advances in hardware have a massive impact on performance. Unless publications explicitly compare different tools, the performance benchmarks are thus not very useful. Even then, the small sample size of the benchmarks means that they may be biased towards some approaches.

All of this means that it is very difficult to determine which approaches are best without comparing all of them thoroughly at the same point in time. This is complicated further by the fact that many of the tools are no longer maintained or were never publically available to begin with.

XIX. Conclusions

The first research question concerns the properties of interest for Datalog-based program analysis tools. The previous sections have examined the expressivity and performance of the various approaches. We have found that expressivity beyond Datalog is needed for many analyses, and that most tools support extensions such as stratified negation. Several tools extend Datalog much further, but the relationships between the different extensions are quite unclear. For performance, properties such as context-sensitivity, parallelization opportunities, and precision seem to be the most important. Surprisingly, some authors found that increased precision can also increase performance by preventing useless computations.

The second research question concerns the approaches used for Datalog-based program analysis. A surprisingly large number of approaches have been tried, though it seems that the combination of translation to relational logic and translation to native code employed in the Soufflé engine is the current state of the art for program analysis when focusing on performance. It is difficult to determine which approach is best from the perspective of expressivity due to the many different extensions in use.

The third research question concerns the comparison of Datalog-based program analysis tools. This question is unfortunately difficult to answer, since most tools differ in expressiveness. A lack of current and statistically significant benchmarks also makes it difficult to determine which approach is best.

A. Recommendations

Of the currently maintained Datalog engines, the Soufflé and LogicBlox engines are probably the best performing for program analyses. The DOOP framework, implemented on top of the Soufflé engine, is probably the best performing Datalog-based analysis framework currently available. If analyses over complete lattices are needed, the IncA and Flix languages may be interesting to consider, though they are less optimized. The IncA language may also be interesting if the objective is real time analysis for use in e.g. an integrated development environment.

When implementing new analyses, one should remember that improved precision may actually improve performance because imprecise analyses may compute large amounts of unnecessary information. It can also pay off to combine multiple analyses to improve precision even further.

Datalog-based tools are still generally best suited for structural analyses such as points-to analysis. For other types of analysis, traditional program analysis tools will probably result in better performance in most cases. In the near future this may change, as there is currently several groups working on extensions to Datalog for lattices and monotone functions.

B. Future research

There are still many unanswered questions and unexplored ideas regarding how Datalog can best be used for program analysis.

An interesting question is whether the various extensions to Datalog form a useful hierarchy of expressivity. If so, it should be possible to classify each tool based on its position in the hierarchy. A unified theory would be very useful to allow interoperation between different tools, since it would make it possible to determine which analyses can be composed. If such a hierarchy exists, it would also be interesting to determine which analyses are supported by each class of extensions. This could also help to determine whether tools that support general monotone frameworks while retaining most of the crispness and simplicity of Datalog are feasible to implement with good performance. It might also be possible to prove complexity results or conduct performance studies about the classes, in which case the trade-offs involved in using the various extensions found today could be more completely understood.

A related avenue of research is specialized evaluation techniques and representation strategies for the various “patterns” found in the various analysis tools available today. Several tools implement specialized datastructures or evaluation techniques for parameters such as sparse versus dense relations, but there could be many more “knobs to tune”. If combined with a hierarchy of extensions to Datalog, it might be possible to determine general guidelines for how good performance can be obtained. The possibility of parallelization would be one of the important factors to consider.

Another area that needs more exploration is extensions and tools for easier development and better usability of analyses. Some of the available tools support syntactic sugar to allow simpler implementation and better readability of analyses without extending the expressivity of the logic. There seems to be a general lack of support for general programming in Datalog, so new tools or embeddings could make it easier to execute analyses and use their results. Integrations with existing tool suites and integrated development environments is most likely a requirement for wide adoption of Datalog-based techniques. General frameworks for composition and integration could be interesting if a hierarchy of Datalog variants can be identified.

References

[1] S. Ceri, G. Gottlob, and L. Tanca, “What you always wanted to know about datalog (and never dared to ask),” *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–66, 1989.

[2] Y. Smaragdakis and G. Balatsouras, “Pointer analysis,” *Foundations and Trends in Programming Languages*, vol. 2, no. 1, pp. 1–80, 2015.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, *Magic sets and other strange ways to implement logic programs (extended abstract)*. Association for Computing Machinery, 1985.

[4] K. T. Tekle and Y. A. Liu, “More efficient datalog queries: Subsumptive tabling beats magic sets,” *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 661–672, 2011.

[5] U. Aßmann, “On edge addition rewrite systems and their relevance to program analysis,” *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1994.

[6] T. W. Reps, “Demand interprocedural program analysis using logic databases,” *Proceedings of the Fifth International Conference on Compiler Construction*, 1994.

[7] R. Wuyts, “Declarative reasoning about the structure of object-oriented systems,” *Proceedings. Technology of Object-oriented Languages. TOOLS 26 (Cat. No. 98ex176)*, pp. 112–24, 1998.

[8] R. Wuyts and S. Ducasse. (2009) Language symbiosis through symbiotic reflection. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.4417

[9] K. Mens, “Automating architectural conformance checking by means of logic meta programming,” Ph.D. dissertation, Vrije Universiteit Brussel, oct 2000.

[10] E. Hajiyev, M. Verbaere, O. D. Moor, and K. D. Volder, “CodeQuest: Querying source code with datalog,” Object-Oriented Programming, Systems, Languages & Applications, 2005.

[11] E. Hajiyev, “Codequest – source code querying with datalog,” Ph.D. dissertation, Oxford University, sep 2005.

[12] E. Hajiyev, M. Verbaere, and O. De Moor, “CodeQuest: Scalable source code queries with datalog,” *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4067 LNCS, pp. 2–27, 2006.

[13] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble, “.QL: object-oriented queries made easy,” *Generative and Transformational Techniques in Software Engineering II. International Summer School, GTTSE 2007. Revised Papers*, pp. 78–133, 2008.

[14] P. Avgustinov, O. De Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented queries on relational data," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 56, pp. 21–225, 2016.

[15] H. Falconer, P. H. Kelly, D. M. Ingram, M. R. Mellor, T. Field, and O. Beckmann, "A declarative framework for analysis and optimization," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4420, pp. 218–232, 2007.

[16] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java, PPPJ 2011*, pp. 71–80, 2011.

[17] M. Madsen, M. H. Yee, and O. Lhoták, "From datalog to FLIX: A declarative language for fixed points on lattices," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 13-17-, no. 6, pp. 194–208, 2016.

[18] G. Balatsouras, K. Ferles, G. Kastrinis, and Y. Smaragdakis, "A datalog model of must-alias analysis," *SOAP 2017 - Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis, Co-located With PLDI 2017*, pp. 7–12, 2017.

[19] N. Grech and Y. Smaragdakis, "P/Taint: Unified points-to and taint analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133926

[20] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of javascript applications in the presence of frameworks and libraries," *2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 - Proceedings*, pp. 499–509, 2013.

[21] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis, "Static analysis of java dynamic proxies," *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 209–220, 2018.

[22] K. De Schutter, "Automated architectural reviews with semmle," *IEEE International Conference on Software Maintenance, ICSM*, p. 6405320, 2012.

[23] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of java enterprise applications: Frameworks and caches, the elephants in the room," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 794–807, 2020.

[24] F. Besson and T. Jensen, "Modular class analysis with DATALOG," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2694, pp. 19–36, 2003.

[25] T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter, "Incrementalizing lattice-based program analyses in datalog," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[26] T. Szabó, E. Kuci, M. Bijman, M. Mezini, and S. Erdweg, "Incremental overload resolution in object-oriented programming languages," *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pp. 27–33, 2018.

[27] M. Maurer, "Holmes: Binary analysis integration through datalog," Ph.D. dissertation, Carnegie Mellon University, Jan 2018. [Online]. Available: https://kilthub.cmu.edu/articles/thesis/Holmes_Binary_Analysis_Integration_Through_Datalog/7571519/1

[28] T. J. Green, M. Aref, and G. Karvounarakis, "LogicBlox, platform and language: A tutorial," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7494, pp. 1–8, 2012.

[29] B. Sarna-Starosta, D. Zook, E. Pasalic, and M. Aref, "Relating constraint handling rules to datalog," *Workshop on Constraint Handling Rules*, 2008.

[30] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: An interactive tutorial," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1213–1216, 2011.

[31] M. Aref, B. Ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the LogicBlox system," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. 2015-, pp. 1371–1382, 2015.

[32] D. Campagna, B. Sarna-Starosta, and T. Schrijvers, "Approximating constraint propagation in datalog," *Colloquium on Implementation of Constraint and LOgic Programming System*, 2011.

[33] A. Bembenek, M. Greenberg, and S. Chong, "Formulog: Datalog for SMT-based static analysis," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, p. 141, 2020.

[34] A. Dura, H. Balldin, and C. Reichenbach, "MetaDL: Analysing datalog in datalog," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 38–43, 2019.

[35] A. Dura and H. Balldin, "MetaDL: Declarative program analysis for the masses," *SPLASH Companion 2019 - Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pp. 17–18, 2019.

[36] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity: The making of a precise and scalable pointer analysis," *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pp. 17–29, 2010.

[37] R. Thiessen and O. Lhoták, "Context transformations for pointer analysis," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 263–277, 2017.

[38] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[39] M. Jeon, S. Jeong, and H. Oh, "Precise and scalable points-to analysis via data-driven context tunneling," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 140 (29 pp.), 2018.

[40] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 38, no. 5, pp. 103–114, 2003.

[41] O. Lhoták and L. Hendren, "Jedd: A BDD-based relational extension of java," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 1, pp. 158–169, 2004.

[42] O. Lhoták, "Program analysis using binary decision diagrams," Ph.D. dissertation, McGill University, Montreal, jan 2006.

[43] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, Oct. 2008. [Online]. Available: https://doi.org/10.1145/1391984.1391987

[44] ——, "Relations as an abstraction for BDD-based program analysis," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 4, Aug. 2008. [Online]. Available: https://doi-org.proxy.findit.dtu.dk/10.1145/1377492.1377494

[45] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 131–144, 2004.

[46] M. S. Lam, J. Whaley, V. Benjamin Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context sensitive program analysis as database queries," *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 1–12, 2005.

[47] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using Datalog with binary decision diagrams for program analysis," *Programming Languages and Systems. Third Asian Symposium, APLAS 2005. Proceedings (Lecture Notes in Computer Science Vol.3780)*, pp. 97–118, 2005.

[48] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva, "Datalog-based program analysis with BES and RWL," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6702, pp. 1–20, 2011.

[49] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA*, pp. 243–261, 2009.

[50] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6702, pp. 245–251, 2011.

[51] M. Bravenboer and Y. Smaragdakis, "Exception analysis and points-to analysis: Better together," *Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA 2009*, pp. 1–11, 2009.

[52] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, "Porting DOOP to soufflé: A tale of inter-engine portability for datalog-based analyses," *SOAP 2017 - Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis, Co-located With PLDI 2017*, pp. 25–30, 2017.

[53] K. Hoder, N. Bjørner, and L. de Moura, "μZ - an efficient engine for fixed points with constraints," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6806, pp. 457–462, 2011.

[54] K. Hoder and N. Bjørner, "Generalized property directed reachability," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7317, pp. 157–171, 2012.

[55] B. S. K. Vorobyov, P. Krishnan, and T. Westmann, "A datalog source-to-source translator for static program analysis: An experience report," *2015*

24th Australasian Software Engineering Conference (ASWEC 2015), pp. 28–37, 2015.

[56] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, “On fast large-scale program analysis in datalog,” *Proceedings of CC 2016: the 25th International Conference on Compiler Construction*, pp. 196–206, 2016.

[57] H. Jordan, B. Scholz, and P. Subotíc, “Soufflé: On synthesis of program analyzers,” *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9780, pp. 422–430, 2016.

[58] H. Jordan, P. Suboti, D. Zhao, and B. Scholz, “Brie: A specialized trie for concurrent datalog,” *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2019*, pp. 31–40, 2019.

[59] P. Subotić, H. Jordan, L. Chang, A. Fekete, and B. Scholz, “Automatic index selection for large-scale datalog computation,” *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 141–153, 2018.

[60] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva, “Static analysis of JAVA programs in a rule–based framework,” *Electronic Notes in Theoretical Computer Science*, 2008.

[61] ——, “Using Datalog and boolean equation systems for program analysis,” *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5596, pp. 215–231, 2009.

[62] ——, “DATALOG_SOLVE: A datalog-based demand-driven program analyzer,” *Electronic Notes in Theoretical Computer Science*, vol. 248, pp. 57–66, 2009.

[63] M. Feliú, C. Joubert, and F. Tarín, “Efficient BES-based bottom-up evaluation of datalog programs,” 2010, allegedly published in Proc. X Jornadas sobre Programacíon y Lengua jes (PROLE 2010).

[64] A. Bembenek and S. Chong, “Formulog: Datalog for static analysis involving logical formulae,” *CoRR*, vol. abs/1809.06274, 2018. [Online]. Available: http://arxiv.org/abs/1809.06274

[65] A. Bembenek, M. Ballantyne, M. Greenberg, and N. Amin, “Datalog-based systems can use incremental SMT solving (extended abstract),” *Electronic Proceedings in Theoretical Computer Science, EPTCS*, vol. 325, 2020.

[66] M. A. Feliú, C. Joubert, and F. Tarín, “Evaluation strategies for datalog-based points-to analysis,” *Electronic Communications of the EASST*, vol. 35, 2010.

[67] M. Madsen and O. Lhoták, “Fixpoints for the masses: Programming with first-class datalog constraints,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, p. 125, 2020.

[68] F. Nielson and H. Seidl. (2001) Succinct solvers. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.582

[69] F. Nielson, H. R. Nielson, and H. Seidl, “A succinct solver for ALFP,” *Nordic Journal of Computing*, vol. 9, pp. 335–372, 2002.

[70] F. Nielson, H. R. Nielson, H. Sun, M. Buchholtz, R. R. Hansen, H. Pilegaard, and H. Seidl, “The succinct solver suite,” *Tools and Algorithms for the Construction and Analysis of Systems. 10th International Conference, TACAS 2004. Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004. Proceedings (lecture Notes in Computer Science Vol.2988)*, pp. 251–65, 2004.

[71] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva, “Implementing datalog in maude,” *Electronic Notes in Theoretical Computer Science*, 2010.

[72] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva, “Defining datalog in rewriting logic,” *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6037, pp. 188–204, 2010.

[73] M. Madsen, M.-H. Yee, and O. Lhoták, “Programming a Dataflow Analysis in Flix,” in *Tools for Automatic Program Analysis (TAPAS)*, 2016.

[74] M. Madsen and O. Lhotak, “Implicit parameters for logic programming,” *PPDP’18: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, pp. 1–14, 2018.

[75] M. Madsen and O. Lhoták, “Safe and sound program analysis with flix,” *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 38–48, 2018.

[76] M. Arntzenius and N. R. Krishnaswami, “Datafun: A functional datalog,” *ICFP 2016 - Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, vol. 51, no. 9, pp. 214–227, 2016.

[77] F. fixed points faster. (2018) Michael arntzenius. [Online]. Available: http://www.rntz.net/files/icfp18hope-finding-fixed-points-faster-abstract.pdf

[78] M. Arntzenius and N. Krishnaswami, “Seminaïve evaluation for a higher-order functional language,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, p. 22, 2020.

[79] T. Szabó, S. Erdweg, and M. Voelter, “IncA: A DSL for the definition of incremental program analyses,” *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 320–331, 2016.

[80] T. Szabó, G. Bergmann, and S. Erdweg, “Incrementalizing inter-procedural program analyseswith recursive aggregation in datalog,” *Second Workshop on Incremental Computing (IC)*, 2019.

[81] T. Szabó, M. Voelter, and S. Erdweg, “IncA$_L$: A DSL for incremental program analysis with lattices,” *First Workshop on Incremental Computing (IC)*, 2017.

[82] W. C. Benton and C. N. Fischer, “Interactive, scalable, declarative program analysis: From prototype to implementation,” *PPDP’07: Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pp. 13–24, 2007.

[83] W. C. Benton, “Fast, effective program analysis for object-level parallelism,” Ph.D. dissertation, University of Wisconsin–Madison, dec 2008.

[84] D. Johnston. (2016) Deductive formulation and implementation of a close-to-source points-to analysis. [Online]. Available: https://github.com/dwtj/deductive-points-to-analysis-paper/blob/master/Deductive%20Close-To-Source%20Points-To%20Analysis.pdf

[85] P. Avgustinov, K. Backhouse, and M. Y. Mo, “Variant analysis with QL,” *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10951, pp. 666–670, 2018.

[86] P. Vilter. (2020) Codebase as database: Turning the IDE inside out with datalog. [Online]. Available: https://petevilter.me/post/datalog-typechecking/

[87] Y. Zhao, G. Chen, C. Liao, and X. Shen, “Towards ontology-based program analysis,” *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 56, pp. 261–2625, 2016.

Appendix A
Search strategy in details

A. Search queries

The following search queries were used:

- Datalog
- program analysis
- Datalog program analysis
- declarative program analysis
- program analysis in Datalog

For each query, the first 100 results on DTU FindIt were considered. Additionally, the first 100 results on Google Search were considered. Each Google Search was conducted with all cookies removed to prevent a “filter bubble” of results relating to each other.

B. References

The references of the sources found during the initial search were considered, and so were the references of any sources found during this process. Most of the sources were found during this process.

C. Searching venues for new publications

Since papers very rarely cite future papers, the above strategy will not necessarily find new research. To combat this, all papers accepted in the last five years were considered for the top five conferences in the field (based on the venues of the existing papers): PLDI, SIGMOD, OOPSLA, ISSTA, and PPDP.

D. Searching author bibliographies with dblp

To further minimize the likelihood that a paper is missed, the dblp database was searched for every author of a relevant paper, and every one of their papers listed there were considered.

Appendix B
Where to look for new developments

The following are lists of the conferences, journals, workshops, and summer schools that have published the papers referenced in this document. Each list is ordered such that the venue with the highest amount of relevant papers comes first. Some of the venues may no longer exist, or be associated with one-off events, but all venues are included for completeness.

A. *Conferences*

- ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)
- International Conference on Management of Data (SIGMOD)
- International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)
- International Symposium on Software Testing and Analysis (ISSTA)
- Symposium on Principles and Practice of Declarative Programming (PPDP)
- European Conference on Object-Oriented Programming (ECOOP)
- International Conference on Computer Aided Verification (CAV)
- Symposium on Principles of Database Systems (PODS)
- Asian Symposium on Programming Languages and Systems (APLAS)
- Spanish Conference on Programming and Computer Languages (PROLE)
- International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)
- International Conference on Functional Programming (ICFP)
- Symposium on Principles of Programming Languages (POPL)
- Colloquium on Implementation of Constraint and LOgic Programming System (CICLOPS)
- International Conference on Software Maintenance (ICSM)
- International Symposium on Formal Methods (FM)
- Australasian Software Engineering Conference (ASWEC)
- International Conference on Compiler Construction (CC)
- International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)
- International Conference on the Principles and Practice of Programming in Java (PPPJ)
- International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)
- International Conference on Theory and Applications of Satisfiability Testing (SAT)

B. *Journals*

- Proceedings of the ACM on Programming Languages (PACMPL)
- Foundations and Trends in Programming Languages
- Electronic Notes in Theoretical Computer Science (ENTCS)
- Electronic Communications of the EASST (ECEASST)

C. *Workshops*

- International Workshop on the State Of the Art in Program Analysis (SOAP)
- International Workshop on Datalog Reloaded
- Workshop on Declarative Program Analysis (DPA)
- Workshop on Tools for Automatic Program Analysis (TAPAS)
- International Workshop on Datalog in Academia and Industry, Datalog 2.0
- International Workshop on Formal Methods for Industrial Critical Systems (FMICS)
- Taller de Programación Funcional (Functional Programming Workshop, TPF)
- Workshop on Automated Verification of Critical Systems (AVoCS)
- Workshop on Higher-Order Programming with Effects (HOPE)
- International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)
- Workshop on Constraint Handling Rules (CHR)

D. *Summer Schools*

- Generative and Transformational Techniques in Software Engineering (GTTSE)

Appendix C
Availability of tools

A. *BDDBDDB*

BDDBDDB is licensed under the LGPLv2 license and is available from http://bddbddb.sourceforge.net/. The project has not been maintained since 2008.

B. *Boolean Equation Systems and Rewriting Logic in Maude*

The DATALOG_SOLVE tool is no longer publicly available. The Datalaude solver is available under a custom license from https://github.com/datalaude/datalaude/. The project has not been maintained since 2009.

C. *Datafun*

Datafun is available to view at https://github.com/rntz/datafun, but no license to use the software is granted. The project has not been maintained since 2020.

D. *DeepWeaver*

The DeepWeaver-1 tool is not publicly available.

E. *DOOP*

DOOP is licensed under the UPL license and is available from https://bitbucket.org/yanniss/doop/. The project is actively maintained.

F. DIMPLE

The DIMPLE framework is not publicly available.

G. Flix

Flix is licensed under the Apache 2.0 license and is available from https://flix.dev/. The project is actively maintained.

H. Formulog

Formulog is licensed under the Apache 2.0 license and is available from https://github.com/HarvardPL/formulog. The project is actively maintained.

I. Holmes

Holmes is licensed under the MIT license and is available from https://github.com/maurer/holmes. The project has not been maintained since 2017.

J. IncA

IncA is licensed under the EPL-2.0 license and is available from https://github.com/szabta89/IncA. The project has not been maintained since 2020.

K. Jedd/PADDLE

Jedd is licensed under the LGPLv2 license and is available from http://www.sable.mcgill.ca/jedd/. The project has not been maintained since 2008.

PADDLE is licensed under the LGPLv2 license and is available from http://www.sable.mcgill.ca/paddle/. The project has not been maintained since 2008.

L. LogicBlox

The LogicBlox solver is available under a commercial license from the Infor company. It may also be available for academic use.

M. MetaDL

MetaDL is licensed under the BSD 2-clause license and is available from https://github.com/lu-cs-sde/metadl. The project has not been maintained since 2019.

N. Semmle

The Semmle solver is available under a commercial license from Github. Github also provides automatic security analyses based on the solver for free.

O. Soufflé

The Soufflé solver is licensed under the UPL license and is available from https://github.com/souffle-lang/souffle. The project is actively maintained.

P. SOUL

SOUL is licensed under the MIT license and is available from http://soft.vub.ac.be/SOUL/. The project does not seem to be maintained.

Q. Succinct solvers

The Succinct solvers are apparently available for academic use, though the license terms are not available. They are available from http://www2.imm.dtu.dk/cs_SuccinctSolver/. The project has not been maintained since 2006.

R. μZ

μZ is licensed under the MIT license. It is available as part of the Z3 solver from https://github.com/Z3Prover/z3.