Program Analysis in Datalog and the Two Language Problem

John Benton
Penn State University
jeb6700@psu.edu

Gang Tan
Penn State University
gxt29@psu.edu

Monica Santra
Penn State University
monikas@psu.edu

***Abstract*—Datalog has recently seen a resurgence in research and industry. Analyses that may take 1000s of lines of code in a language like C can be replicated in 10s of lines of Datalog code. However, Datalog programs require a set of facts to work from. In a low level project like program analysis, these facts can be difficult to generate. Designing a program analysis framework for Datalog allows for programmers to stop worrying about generating low level information and just write analyses but pure Datalog may not be the right tool for the job.**

***Index Terms*—program analysis, datalog**

I. Introduction

This paper recounts our attempt at designing a general binary analysis framework for the Datalog programming language, specifically Souffle. We introduce the Datalog Analysis Tool (DAT), designed to allow programmers to stop thinking about low level concepts and reason about programs at a high level using Datalog rules. By using the Binary Analysis Program (BAP) we were able to reason about low level aspects of a program and formulate that information into Datalog facts to perform analysis. Section II describes background information necessary for understanding the design of this framework, which is detailed in Section III. Section IV evaluates the design and focuses on the pitfalls of using pure Datalog for program analysis. We conclude with a general overview of our findings as well as recommendations for future research.

II. Background and Related Work

A. Datalog Program Analysis

Datalog is a logical, declarative query language that can efficiently specify static analyses. A Datalog program is made up of a set of rules that takes input from a set of facts and uses fixpoint computation to create a set of results. Datalog has a history of being used for static analyses such as points-to analysis[1], [2] as well as other binary analysis endeavors like disassembly[3] and dead code analysis[4]. Datalog has consistently been able to represent complex program analysis algorithms in significantly less code than other popular languages.

B. BAP

The Binary Analysis Program (BAP) designed by researchers at Carnegie Mellon is a framework for writing binary analyses[5]. The BAP API uses an intermediate language (IL) to expose aspects of the program being analyzed. Conceptually, BAP allows the programmer to view a program as a set of subroutines, a subroutine as a set of basic blocks, and basic blocks as being made up specific instructions. This modularization makes writing analyses in BAP well structured and the type safety of OCaml ensures that you will never mistake one term for another, making BAP an obvious choice for generating the low level facts of the DAT framework.

III. Design

In designing DAT, the biggest question we asked is how much information to encode into Datalog facts. We decided to iteratively add features in order to replicate popular analyses.

First, we replicated the BAP tutorial: checking whether a function can ever occur after another.

The BAP API allows for programs to be turned into a directed graph where every edge is a function call. Encoding these edges into Datalog rules allows the DAT framework to reason about the callgraph of a program. We can conceptualize the check path problem using Linear Temporal Logic:

$$f \to G\ not(g)$$

where f and g are propositions that are true if a call to a the corresponding function occurred, and $G\ x$ is the LTL operator that requires the predicate x to be true on the entire subsequent path.

This fits very easily into the Datalog rules:

```
path(src, dst) :- calls(src, dst).
path(src, dst) :- path(src, x),
calls(x, dst).

check_path(src, dst) :- path(src, dst).
check_path(src, dst) :- path(p, src),
path(q, dst), path(p, q).
```

Where calls(src, dst) references the encoded callgraph and is done in only 8 lines of OCaml.

The BAP implementation requires a 4 line function to make sure both f and g exist, a 2 line function to check whether there is a direct path between them, a 9 line function to check the control flow graph for a possible call, and an 11 line function to run them all together in the correct order. Obviously some of it is done to show off all of BAP features, but clearly Datalog is a more efficient representation of the logic.

Next, we replicated deadcode analysis. BAP makes this easier by doing static single assignment (SSA) which is where every variable is defined exactly once. In situations where control flow allows for a variable to have more than one possible value (an if-else statement) phi-nodes are used to denote the possible values a single variable can hold. The example:

```
x = 3
if x > 10:
    x = 15
else:
    x = 20
```

would be represented as:
$x_1 = 3$
$x_2 = 15$
$x_3 = 20$
$x_4 = \Phi(x_2, x_3)$

By encoding every definition and use of the variables in SSA into a Datalog rule, the DAT framework is able to trivially perform deadcode analysis, as any variable that has a definition but not a use rule is not used.

IV. Evaluation

Initial attempts at using the DAT framework for analysis were promising, as it was able to recreate the BAP tutorial (checking whether a call to a function occurs after another call) in significantly less code. However, while trying to replicate more complex analyses we ran into problems in both our design strategy and with Datalog itself.

A. Design Strategy Problems

We started to create this framework by replicating popular analyses in Datalog until DAT eventually became a full framework. However, this modularity meant adding new features often felt like designing an entirely new framework. We believe a better approach would have been to focus strictly on encoding the callgraph and control-flow graph into Datalog rules first, before then refining our framework as needed.

B. The Two Language Problem

The Two Language Problem occurs when researchers want to develop a tool in a high level language for better understanding but then implement it in a lower level for more speed[6]. So called ”pure” Datalog, where the language only supports fixpoint computation on database rules, presents an interesting variant on this problem. In pure Datalog, all information must be formatted in to these databases before any analysis is started. Thus, any Datalog analysis must begin with a language that isn't Datalog to encode these rules. Furthermore, another language may be required after the analysis is complete for post-processing.

The question that naturally follows is when to switch over but when doing program analysis, that question is much more nuanced than it initially appears. Switching over early may seem like a good option but then any analysis is forced to encode a substantial amount of low level information into Datalog rules which can cause projects to quickly become logically convoluted very quickly.

Switching over late leads to a different problem: why ever switch at all? Conceptually, program analysis in Datalog can be thought of as pre-processing a program into initial rules, running the Datalog analysis, and then doing post-processing on the resulting fact files to actually implement that analysis. Importantly, the pre/post processing will generally be done in the same language for ease of implementation. At a certain point, it becomes much easier to do the entire analysis in a single language than worry about pre/post processing data.

Any framework written in pure Datalog will have to perfectly balance that line, a very difficult task.

C. Let's Go Impure

If doing program analysis in Datalog is a language flexibility problem, the solution is to create a new language.

A general purpose language that supports Datalog is much more flexible than pure Datalog. Even though some pre/post processing is still necessary, it can all be done in the same language, creating a better development experience. Languages like Flix[7] and Flan[8] currently exist to address this problem but are still early in development.

V. Conclusion

Datalog is incredibly powerful and certainly has a place in program analysis. But a programming language that needs another to start and finish its task is not one that is suited for production-level development. A good program analysis framework requires a flexible language which is not currently something Datalog offers. Further research is necessary in order to make a general purpose language that supports Datalog with optimal execution. Until then, excitement about Datalog will continue to be exclusive to academia.

References

[1] S. H. Kim, C. Sun, D. Zeng, and G. Tan, “Refining indirect call targets at the binary level,” in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[2] Y. Smaragdakis and M. Bravenboer, “Using datalog for fast and easy program analysis,” vol. 6702, Jan. 2010, pp. 245–251, ISBN: 978-3-642-24205-2. DOI: 10.1007/978-3-642-24206-9_14.

[3] A. Flores-Montoya and E. Schulte, “Datalog disassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 1075–1092, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya.

[4] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, “Using datalog with binary decision diagrams for program analysis,” in *Programming Languages and Systems*, K. Yi, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 97–118, ISBN: 978-3-540-32247-4.

[5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “Bap: A binary analysis platform,” in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV’11, Snowbird, UT: Springer-Verlag, 2011, pp. 463–469, ISBN: 9783642221095.

[6] J. Bezanson, “Abstraction in technical computing,” Ph.D. dissertation, Massachusetts Institute of Technology, Jan. 2015.

[7] M. Madsen, M.-H. Yee, and O. Lhoták, “From datalog to flix: A declarative language for fixed points on lattices,” *SIGPLAN Not.*, vol. 51, no. 6, pp. 194–208, Jun. 2016, ISSN: 0362-1340. DOI: 10.1145/2980983.2908096. [Online]. Available: https://doi.org/10.1145/2980983.2908096.

[8] S. Abeysinghe, A. Xhebraj, and T. Rompf, “Flan: An expressive and efficient datalog compiler for program analysis,” *Proc. ACM Program. Lang.*, vol. 8, no. POPL, Jan. 2024. DOI: 10.1145/3632928. [Online]. Available: https://doi.org/10.1145/3632928.