Isabelle-verified correctness of Datalog programs for program analysis

Anders Schlichtkrull
Aalborg University
Copenhagen, Denmark
andsch@cs.aau.dk

René Rydhof Hansen
Aalborg University
Aalborg, Denmark
rrh@cs.aau.dk

Flemming Nielson
Technical University of Denmark
Kongens Lyngby, Denmark
fnie@dtu.dk

ABSTRACT

Static program analysis has become an essential tool for developers to find and avoid bugs as well as security vulnerabilities. This is particularly important for applications requiring formal verification, e.g., safety- or security-critical applications.

We formalize and prove correct all the components needed to specify and perform static analysis based on the *Datalog* logic programming language, including the first known Isabelle formalization of *stratified Datalog*. In addition the existence of *least solutions* for any stratified Datalog program is established and proven. We demonstrate the usefulness of our formalization by further formalizing the general *Bit-Vector Framework* and prove correct four classic analyses in this framework.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; **Program analysis**.

KEYWORDS

Datalog, static analysis, formal methods, Isabelle

ACM Reference Format:
Anders Schlichtkrull, René Rydhof Hansen, and Flemming Nielson. 2024. Isabelle-verified correctness of Datalog programs for program analysis. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3605098.3636091

1 INTRODUCTION

The ability of static analysis to provide automated and scalable bug-finding and verification has made it an intrinsic part of the development process, not least for critical applications requiring high assurance levels. Despite this, relatively little work has been published on formalizing and mechanically verifying static analyses and the frameworks for designing and implementing such analyses, leaving a "formalization gap" where applications may be formally verified by analysis tools and frameworks that are not.

In this work we use Isabelle [7, 8] to formalize and develop a theory for *stratified Datalog* [2], a non-Turing complete logic programming language that has been used to specify and implement static analyses in a succinct form that scales to real-life code bases. To the best of our knowledge this is the first such formalization in Isabelle. We use our formalization to prove that all analyses in the so-called *Bit-Vector Framework* [6] can be encoded as corresponding Datalog analyses. The Bit-Vector Framework, sometimes known as gen/kill analysis or simply bit-vector data flow analysis, is a general and widely used subset of Kildall's classic data flow analysis framework [4] and the more general *monotone framework* [3], consisting of the program analyses that can be defined and implemented using efficient bit-vector representations of the data flow equations and operations. To the best of our knowledge, this is the first published proof that Datalog subsumes the Bit-Vector Framework. We further use the Datalog formalization to formally establish the existence of *least solutions* for any stratified Datalog program, essential for reasoning about negations in clauses and for obtaining the most precise analysis results. This is a non-trivial task, since the ordering of solutions, and hence the least solution, has to take stratification into account.

Lastly we illustrate the usefulness and applicability of our Datalog and Bit-Vector Framework formalization by formalizing and proving correct four classic bit-vector analyses, i.e., instances of the Bit-Vector Framework. Analyses formulated as Datalog programs are of particular interest because they have been shown to scale to large code bases and are often easier and less time consuming to define and develop [11–13].

In summary, our contributions are: *(1)* a formalization of stratified Datalog, *(2)* a formal proof that a *least* solution exists for any stratified Datalog program, *(3)* a formalization of program graphs, *(4)* a formal proof that all bit-vector analyses can correctly be formulated as Datalog programs, *(5)* a formalization and correctness proof of the reaching definitions, live variables, available expressions, and very busy expressions analyses. Our work is available online [10].

2 DATALOG FOR PROGRAM ANALYSIS

Program analysis is traditionally concerned with proving properties about programs at compile time, e.g., to enable compiler optimisations or guaranteeing the absence of bugs. There are numerous approaches to defining program analyses but the work here is based on a recent approach using the *Datalog* language for specification and implementation of the analyses [5, 6, 11, 12].

Datalog is a logic programming language inspired by and resembling (a simplified subset of) Prolog that is *not* Turing complete [2]. This choice facilitates developing program analyses in a way that separates specification and implementation of the analysis while leveraging efficient implementation that scales to real-life code bases.

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3636091

Using Datalog for program analysis generally works by defining Datalog predicates that track analysis information for each program point and Datalog clauses that propagate this information according to the instructions of the program under analysis. For typical analyses, this specification only has to be done once for each *type* of instruction, e.g., all assignments are handled by similar Datalog clauses. The Datalog clauses comprising the analysis specification can then be used to perform analysis of a concrete program by first generating a set of *ground facts* that represents the program to be analyzed and then applying the Datalog program to the ground facts. In particular, the *control flow* is encoded as ground facts, e.g., by encoding the *program graph* of the program to be analyzed as shown in this paper. The *solution* to the Datalog program is now also the analysis result.

3 FORMALIZING DATALOG IN ISABELLE/HOL

Given the widespread use of Datalog in diverse areas of application our formalization is of separate interest also outside of program analysis. The goal is a formalization enabling us to not only establish meta-theoretical results about Datalog, but to prove correctness of specific Datalog programs. This is in contrast to the formalization by Benzaken, Contejean and Bembrava [1] whose goal is to prove the correctness of strategies for evaluating clauses of Datalog programs.

For convenience in specifying analyses, we use a Datalog extension, called *stratified Datalog*, that allows negation. To ensure well-definedness, negation must be used in a structured way to avoid circular, nonsensical clauses. While convenient for users, stratification makes the formalization more involved and demanding. Due to the semantics and the ordering of solutions to Datalog programs, that needs to take stratification into account, we also need to prove that *minimal* solutions to stratified Datalog programs coincide with *least* solutions. We prove this as a consequence of a proof of the existence of a least solution (also known as the perfect model [2, 9]).

Our formalization of Datalog is based on Nielson and Nielson's textbook [6] and was done using Isabelle/HOL [7, 8] which is a proof assistant for Higher-Order Logic. Proof assistants allow their users to define objects from mathematics, logic and computer science and to prove lemmas and theorems about them. The proof assistant checks if the proofs are correct and can also do parts of proofs automatically. The advantage is clear: a proof done in a proof assistant is highly trustworthy. Proof assistants typically rely on a small, trusted kernel based on a relatively simple proof system. Isabelle/HOL's logic (HOL) can be seen as a combination of typed functional programming with logic. Therefore a formalization will specify the *types* of objects we want to reason about, and then we study *terms* (representing mathematical objects) of these types.

To give a flavour of the actual formalisation we present the following excerpt formalizing the semantics of Datalog. For lack of space, we do not go into any details:

fun eval_id ($[\![\_]\!]_{\mathsf{id}}$) **where**
  $[\![\mathsf{Var}\ x]\!]_{\mathsf{id}}\ \sigma = \sigma\ x$
| $[\![\mathsf{Cst}\ c]\!]_{\mathsf{id}}\ \sigma = c$

fun eval_ids ($[\![\_]\!]_{\mathsf{ids}}$) **where**
  $[\![ids]\!]_{\mathsf{ids}}\ \sigma = \mathsf{map}\ (\lambda a.\ [\![a]\!]_{\mathsf{id}}\ \sigma)\ ids$

fun meaning_lh ($[\![\_]\!]_{\mathsf{lh}}$) **where**
  $[\![p(ids)]\!]_{\mathsf{lh}}\ \varrho\ \sigma \longleftrightarrow [\![ids]\!]_{\mathsf{ids}}\ \sigma \in \varrho\ p$

fun meaning_rh ($[\![\_]\!]_{\mathsf{rh}}$)**where**
  $[\![a \mathbin{\boldsymbol{=}} a']\!]_{\mathsf{rh}}\ \varrho\ \sigma \longleftrightarrow [\![a]\!]_{\mathsf{id}}\ \sigma = [\![a']\!]_{\mathsf{id}}\ \sigma$
| $[\![a \mathbin{\boldsymbol{\neq}} a']\!]_{\mathsf{rh}}\ \varrho\ \sigma \longleftrightarrow [\![a]\!]_{\mathsf{id}}\ \sigma \neq [\![a']\!]_{\mathsf{id}}\ \sigma$
| $[\![p(ids)]\!]_{\mathsf{rh}}\ \varrho\ \sigma \longleftrightarrow [\![ids]\!]_{\mathsf{ids}}\ \sigma \in \varrho\ p$
| $[\![\boldsymbol{\neg} p(ids)]\!]_{\mathsf{rh}}\ \varrho\ \sigma \longleftrightarrow \neg\ [\![ids]\!]_{\mathsf{ids}}\ \sigma \in \varrho\ p$

fun meaning_rhs ($[\![\_]\!]_{\mathsf{rh}}$) **where**
  $[\![rhs]\!]_{\mathsf{rhs}}\ \varrho\ \sigma \longleftrightarrow (\forall rh \in \mathsf{set}\ rhs.\ [\![rh]\!]_{\mathsf{rh}}\ \varrho\ \sigma)$

fun meaning_cls ($[\![\_]\!]_{\mathsf{cls}}$) **where**
  $[\![p(ids) \leftarrow rhs]\!]_{\mathsf{cls}}\ \varrho\ \sigma \longleftrightarrow$
    $([\![rhs]\!]_{\mathsf{rhs}}\ \varrho\ \sigma \longrightarrow [\![p(ids)]\!]_{\mathsf{lh}}\ \varrho\ \sigma)$

---

fun solves_lh ($\models_{\mathsf{lh}}$) **where** $\varrho \models_{\mathsf{lh}} lh \longleftrightarrow (\forall \sigma.\ [\![lh]\!]_{\mathsf{lh}}\ \varrho\ \sigma)$

fun solves_rh ($\models_{\mathsf{rh}}$) **where** $\varrho \models_{\mathsf{rh}} rh \longleftrightarrow (\forall \sigma.\ [\![rh]\!]_{\mathsf{rh}}\ \varrho\ \sigma)$

definition solves_cls ($\models_{\mathsf{cls}}$) **where**
  $\varrho \models_{\mathsf{cls}} c \longleftrightarrow (\forall \sigma.\ [\![c]\!]_{\mathsf{cls}}\ \varrho\ \sigma)$

definition solves_program ($\models_{\mathsf{dl}}$) **where**
  $\varrho \models_{\mathsf{dl}} dl \longleftrightarrow (\forall cls \in dl.\ \varrho \models_{\mathsf{cls}} cls)$

REFERENCES

[1] Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. 2017. Certifying Standard and Stratified Datalog Inference Engines in SSReflect. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10499)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer, 171–188. https://doi.org/10.1007/978-3-319-66107-0_12

[2] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166. https://doi.org/10.1109/69.43410

[3] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica* 7 (1977), 305–317.

[4] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL 1973)*. 194–206. https://doi.org/10.1145/512927.512945

[5] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. 194–208.

[6] Flemming Nielson and Hanne Riis Nielson. 2020. Program Analysis (an Appetizer). *CoRR* abs/2012.10086 (2020). arXiv:2012.10086 https://arxiv.org/abs/2012.10086

[7] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics - With Isabelle/HOL*. Springer. https://doi.org/10.1007/978-3-319-10542-0

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[9] Teodor C. Przymusinski. 1988. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, Jack Minker (Ed.). Morgan Kaufmann, 193–216. https://doi.org/10.1016/b978-0-934613-40-8.50009-9

[10] Anders Schlichtkrull, René Rydhof Hansen, and Flemming Nielson. 2023. Formal proof development for the present paper. https://github.com/anderssch/LTS-formalization/tree/SAC2023/Datalog.

[11] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded - Revised Selected Papers of the First International Workshop on Datalog (Datalog 2010) (Lecture Notes in Computer Science, Vol. 6702)*. Springer, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[12] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS 2005) (Lecture Notes in Computer Science, Vol. 3780)*. Springer, 97–118. https://doi.org/10.1007/11575467_8

[13] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004)*. ACM, 131–144. https://doi.org/10.1145/996841.996859