



# Making Formulog Fast: An Argument for Unconventional Datalog Evaluation

AARON BEMBENEK, University of Melbourne, Australia

MICHAEL GREENBERG, Stevens Institute of Technology, USA

STEPHEN CHONG, Harvard University, USA

With its combination of Datalog, SMT solving, and functional programming, the language Formulog provides an appealing mix of features for implementing SMT-based static analyses (e.g., refinement type checking, symbolic execution) in a natural, declarative way. At the same time, the performance of its custom Datalog solver can be an impediment to using Formulog beyond prototyping—a common problem for Datalog variants that aspire to solve large problem instances. In this work we speed up Formulog evaluation, with some surprising results: while 2.2× speedups can be obtained by using the conventional techniques for high-performance Datalog (e.g., compilation, specialized data structures), the big wins come by abandoning the central assumption in modern performant Datalog engines, semi-naïve Datalog evaluation. In the place of semi-naïve evaluation, we develop eager evaluation, a concurrent Datalog evaluation algorithm that explores the logical inference space via a depth-first traversal order. In practice, eager evaluation leads to an advantageous distribution of Formulog’s SMT workload to external SMT solvers and improved SMT solving times: our eager evaluation extensions to the Formulog interpreter and Soufflé’s code generator achieve mean 5.2× and 7.6× speedups, respectively, over the optimized code generated by off-the-shelf Soufflé on SMT-heavy Formulog benchmarks.

All in all, using compilation and eager evaluation (as appropriate), Formulog implementations of refinement type checking, bottom-up pointer analysis, and symbolic execution achieve speedups on 20 out of 23 benchmarks over previously published, hand-tuned analyses written in F#, Java, and C++, providing strong evidence that Formulog can be the basis of a realistic platform for SMT-based static analysis. Moreover, our experience adds nuance to the conventional wisdom that traditional semi-naïve evaluation is the one-size-fits-all best Datalog evaluation algorithm for static analysis workloads.

CCS Concepts: • **Software and its engineering** → **Constraint and logic languages**; *Automated static analysis*; • **Computing methodologies** → *Shared memory algorithms*.

Additional Key Words and Phrases: Datalog, SMT solving, Formulog, compilation, parallel evaluation

## ACM Reference Format:

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2024. Making Formulog Fast: An Argument for Unconventional Datalog Evaluation. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 314 (October 2024), 30 pages. <https://doi.org/10.1145/3689754>

## 1 Introduction

Through combining Datalog, SMT solving, and functional programming, Formulog [Bembenek et al. 2020b] aims to be a domain-specific language for declaratively implementing SMT-based static analyses such as refinement type checking and symbolic execution. Formulog-based analyses achieve speedups over (non-Datalog) reference implementations on several case studies thanks to

---

Authors’ Contact Information: Aaron Bembenek, University of Melbourne, Parkville, Australia, [aaron.bembenek@unimelb.edu.au](mailto:aaron.bembenek@unimelb.edu.au); Michael Greenberg, Stevens Institute of Technology, Hoboken, USA, [michael@greenberg.science](mailto:michael@greenberg.science); Stephen Chong, Harvard University, Cambridge, USA, [chong@seas.harvard.edu](mailto:chong@seas.harvard.edu).

---



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2024 Copyright held by the owner/author(s).

ACM 2475-1421/2024/10-ART314

<https://doi.org/10.1145/3689754>

high-level optimizations like automatic parallelization and goal-directed evaluation; nonetheless, the prototype Formulog interpreter is not a heavily optimized system, and on pure Datalog programs it can be  $\sim 7\times$  slower than the industrial-strength Datalog solver Soufflé [Jordan et al. 2016; Scholz et al. 2016]. This performance gap is hardly surprising, as building a performant Datalog system is a challenging endeavor—involving multicore programming, specialized data structures, and subtle engineering—and Soufflé has benefited from multiple person-years of engineering investment, as well as industry support.

The engineering gap between Formulog and an optimized Datalog system like Soufflé has limited Formulog’s use in practice. For example, Smaragdakis et al. [2021] conclude that Formulog’s custom Datalog engine will not scale to the problems targeted by their novel “symvalic” analysis; this is despite the fact that Formulog has the right language features, and “an evolved implementation [of Formulog] that will seamlessly combine Datalog rules and symbolic reasoning with high performance will be an ideal platform for symvalic analysis in the future.” Formulog may have been a success from the perspective of language design, but its potential is only partially realized if the performance of its prototype implementation prevents interested clients from using it.

Moreover, performance concerns do not apply solely to Formulog: indeed, any novel Datalog variant that aspires to solve industrial-size problem instances faces the challenge of closing the sizable gap between research artifact and high-performance system.

Fortunately, over the past decade, a recipe has emerged for fast and scalable Datalog evaluation, based on techniques pioneered by Soufflé: compile the Datalog program to imperative code that specializes the semi-naïve Datalog evaluation algorithm [Bancilhon 1986] to that source program [Jordan et al. 2016; Scholz et al. 2016], and link in optimized data structures [Jordan et al. 2019a,b, 2022; Subotić et al. 2018]. This is the general approach taken by recent high-performance Datalog engines (beyond Soufflé) that target static analysis applications. For example, Ascent [Sahebolamri et al. 2022] uses macros to compile embedded Datalog code to Rust code implementing semi-naïve evaluation for that program, and then BYODS [Sahebolamri et al. 2023] adds the ability for users to link in custom data structures ideal for the particular workload; Flan [Abeysinghe et al. 2024] uses a metaprogramming framework to specialize both a semi-naïve Datalog interpreter and the data structures used by the interpreter to the Datalog program being evaluated.

Pacak and Erdweg [2022] sensibly propose “frontend compilation” as an approach to making novel Datalog-inspired languages performant: compile them to existing high-performance Datalog engines that already implement the traditional recipe for fast and scalable Datalog evaluation. They contrast frontend compilation with two existing classes of approaches to making Datalog systems more expressive or usable: “frontend-first” approaches, which start by adding language features to Datalog and then build a custom backend to support them (here they explicitly use Formulog as an example), and “backend-first” approaches, which start with existing Datalog infrastructure and add new features on top of it. They argue that frontend compilation achieves the best of both worlds, by allowing expressive frontend design, while reusing existing Datalog infrastructure.

Our first attempt to speed up Formulog evaluation follows the frontend compilation approach of Pacak and Erdweg [2022]. We develop a compiler from the Datalog fragment of Formulog to Soufflé, which in turn generates C++ code; this code is linked with the C++ code generated directly by our compiler for the non-Datalog features of Formulog (the functional fragment and SMT runtime). The generated code is arithmetic mean  $2.2\times$  faster than the baseline Formulog interpreter on Formulog benchmarks (min / median / max:  $1.0\times / 1.7\times / 4.3\times$ ); the approach works particularly well for programs (like a bottom-up Java pointer analysis [Feng et al. 2015]) where most computational work happens in the Datalog fragment (arith mean / min / median / max:  $3.2\times / 1.3\times / 3.4\times / 4.3\times$ ).

Given the engineering effort put into Soufflé, this speedup is not surprising; what is surprising is that one can do better, with little engineering effort. With the addition of just 200 lines of code, the prototype Formulog interpreter beats the optimized C++ code generated by Soufflé with an arithmetic mean  $5.2\times$  speedup on non-trivial SMT-heavy benchmarks (min / median / max:  $0.41\times / 1.2\times / 31\times$ ), such as sophisticated refinement type checking [Bierman et al. 2012] and KLEE-style symbolic execution [Cadar et al. 2008], where there are up to hundreds of thousands of SMT calls and tens of millions of derived tuples. The code generated by Soufflé has many potential advantages over the Formulog interpreter (which is written in Java and uses relatively naive data structures); the interpreter’s trick is to abandon one of the key assumptions underlying modern high-performance Datalog engines, semi-naive evaluation.

Semi-naive evaluation [Bancilhon 1986] has been the standard Datalog evaluation algorithm for decades, and is the backbone of virtually all modern Datalog engines (especially those that target static analysis applications) [Ketsman and Koutris 2022]. Semi-naive evaluation cuts down on the number of redundant inferences made while evaluating Datalog rules, but it has a crucial limitation when it comes to Formulog evaluation: it enforces a breadth-first search (BFS) over the logical inference space, deriving inferences of proof height one, followed by inferences of proof height two, three, etc. In the context of Formulog, the order of logical inferences determines how the SMT queries that arise in the course of Datalog rule evaluation are distributed amongst external SMT solvers; thus, improved Formulog performance can be achieved by making inferences in an order that leads to faster external SMT solving (whether thanks to improved opportunities for incremental SMT solving [Bembenek et al. 2020a; Eén and Sörensson 2003] or just a more favorable distribution of the SMT workload). Here, the BFS of semi-naive evaluation can be suboptimal.

In place of traditional semi-naive evaluation, we develop *eager evaluation*, a Datalog evaluation algorithm that uses work-stealing-based parallelism [Arora et al. 1998; Blumofe and Leiserson 1999; Mohr et al. 1990] to perform a quasi-depth-first search (DFS) of the logical inference space.<sup>1</sup> Instead of batching work into explicit rounds of evaluation (as in semi-naive evaluation), eager evaluation eagerly pursues the consequences of the most recent derivations. While this lack of batching means that eager evaluation can sometimes make more redundant derivations than semi-naive evaluation, in practice it is a good evaluation algorithm for Formulog: eager evaluation leads to advantageous distributions of the SMT workload across external SMT solvers and thus less time spent in SMT solving, a key determiner of overall performance for SMT-heavy Formulog programs.

We extend Soufflé with proof-of-concept support for generating C++ code that performs eager evaluation. On SMT-heavy benchmarks, the generated code achieves an arithmetic mean  $1.8\times$  speedup (min / median / max:  $0.93\times / 1.3\times / 4.1\times$ ) over the Formulog interpreter running in eager evaluation mode. This represents an arithmetic mean  $8.8\times$  speedup over the baseline Formulog interpreter (min / median / max:  $0.73\times / 4.8\times / 38\times$ ). As our modifications to the Soufflé codebase amount to merely 500 additional lines of code, our experience demonstrates the feasibility of using existing infrastructure to build Datalog engines that perform eager evaluation. This is a key strength of the algorithm, as a novel Datalog evaluation technique is unlikely to be used in practice if it is incompatible with the substantial research and sophisticated, labor-intensive engineering that has gone into making existing Datalog systems like Soufflé performant.

Overall, our compiler from Formulog to Soufflé—set to use our eager evaluation extension for SMT-heavy benchmarks—provides an arithmetic mean speedup of  $6.4\times$  over the baseline Formulog interpreter (min / median / max:  $0.74\times / 3.4\times / 38\times$ ; see summary in Table 1), and beats all interpreter modes on 22 out of 23 benchmarks. This translates to competitive performance relative to previously

<sup>1</sup>The original Formulog paper [Bembenek et al. 2020b] uses a preliminary version of eager evaluation for one of the case studies; however, it does not describe it nor evaluate it with respect to semi-naive evaluation. This paper fills those gaps.

Table 1. Compilation and eager evaluation lead to substantial arithmetic mean speedups over the baseline Formulog interpreter. The headings “+eager (interpret)” and “+eager (compile)” denote using interpreted and compiled eager evaluation modes, respectively, for SMT-heavy benchmarks, and the base compiler (to off-the-shelf Soufflé) for the other benchmarks. Eager evaluation is key to getting top performance on SMT-heavy benchmarks, where *interpreted* eager evaluation actually beats *compiled* semi-naive code.

Benchmarks	Base compiler	+eager (interpret)	+eager (compile)
SMT-heavy (13)	1.5×	5.7×	8.8×
All (23)	2.2×	4.6×	6.4×

published, hand-tuned analyses written in F<sup>#</sup>, Java, and C++: on 20 out of 23 benchmarks, some Formulog mode is the fastest; on synthetic symbolic execution benchmarks, compiled Formulog’s eager evaluation mode achieves an arithmetic mean 100× speedup (min/median/max: 7.0×/12×/870×) over the symbolic execution tool KLEE [Cadaru et al. 2008]. These results provide strong evidence that Formulog can be the basis of a realistic platform for SMT-based static analysis.

### 1.1 Impact Beyond Formulog

While we focus on Formulog, our work has broader impact along three main dimensions.

*We push the frontiers of Datalog for static analysis.* Our work helps redefine the boundary of what is known to be feasible for Datalog-based static analysis, an active area of PL research that strives to make it possible to write analyses that are declarative (i.e., at the level of mathematical specifications) and can be efficiently evaluated. The original Formulog paper [Bembenek et al. 2020b] primarily answers an expressivity-focused research question: “Can SMT-based analyses like symbolic execution and refinement type checking be naturally expressed in a Datalog-like language?” Our paper answers equally important performance-focused research questions: “Can SMT-based analyses like symbolic execution and refinement type checking be efficiently evaluated in a Datalog-like language? Are existing ways of speeding up Datalog evaluation sufficient, or are new approaches necessary?” In answering these questions—and by proposing a targeted evaluation algorithm that can be easily integrated with existing high-performance Datalog infrastructure—we give good evidence for the practicality of developing SAT/SMT-based analyses in Datalog, a research direction that is of interest to other (non-Formulog) declarative static analysis projects [Abeysinghe et al. 2024; Aiken et al. 2007; Smaragdakis et al. 2021].

*We highlight how Datalog variants can be sensitive to logical inference order.* Other Datalog variants that interact with stateful external systems might benefit from evaluation modes, like eager evaluation, that explore the space of logical inferences in an order that is advantageous from the perspective of those external systems. For example, a Datalog system that interacts with an external database that does not fit in memory would want to avoid inducing irregular database access patterns that lead to disk thrashing; a Datalog system that interacts with AI chatbots (cf. Vieira [Li et al. 2024]) might get better quality responses if the sequence of prompts sent to a chatbot forms a coherent line of conversation.

*We question the conventional wisdom on speeding up Datalog.* Our experiences add some nuance to the conventional narrative of how to make non-distributed Datalog variants fast. To this point, the suggestion has been that the combination of compilation, specialized data structures, and semi-naive evaluation is sufficient (perhaps with some additional attention to join orders and join algorithms). Our work demonstrates that this mindset runs the risk of missing out on performance improvements

coming from the choice of evaluation algorithm itself, and that the “frontend compilation” approach of Pacak and Erdweg [2022]—while a reasonable starting point—can ultimately provide only limited performance gains if existing Datalog systems do not anticipate the potentially idiosyncratic workloads of the Datalog variant being compiled. Over time, Datalog has moved wildly beyond its origins as a database query language, with applications in such diverse domains as static analysis [Bravenboer and Smaragdakis 2009; Flores-Montoya and Schulte 2020; Grech et al. 2019, 2018; Smaragdakis et al. 2021; Szabó et al. 2018, 2021; Tsankov et al. 2018; Whaley et al. 2005; Whaley and Lam 2004], networking [Loo et al. 2006; Ryzhyk and Budiu 2019], distributed systems [Alvaro et al. 2010a,b], access control [Dougherty et al. 2006; Li and Mitchell 2003], big data analytics [Seo et al. 2013; Shkapsky et al. 2016], and neurosymbolic AI [Huang et al. 2021; Li et al. 2024, 2023]. As the applications of Datalog continue to evolve—whether to entirely new domains, or simply to new forms in existing domains (e.g., SMT-based static analyses)—we should keep in mind the possibility of developing novel Datalog evaluation algorithms to better fit new workloads. A benefit of a declarative language like Datalog is that it is amenable to many different evaluation techniques; our work demonstrates one way that Datalog system designers can take advantage of this flexibility.

## 1.2 Contributions

At a high level, our paper explores how to efficiently execute SMT-based program analyses like symbolic execution and refinement type checking in a Datalog-like language. The Formulog versions of these analyses differ from the types of analyses traditionally written in Datalog, in form (due to the extensive use of functional code), in the analysis logic they implement (thanks to SMT solving), and in their workloads (as SMT solving can be the most computationally intensive part). By developing practical techniques to speed up these Datalog-based analyses, and showing that the Formulog versions can be competitive with—and sometimes much faster than—non-Datalog versions implemented in conventional languages, our work contributes to the larger research project of declarative, Datalog-based static analysis. More concretely, we make the following contributions:

- We design and evaluate a compiler from Formulog to off-the-shelf Soufflé (Section 3). On the Formulog benchmark suite, the code produced by the compiler achieves an arithmetic mean  $2.2\times$  speedup over the Formulog interpreter’s semi-naïve evaluation mode. Through this compiler from Formulog to Soufflé, we evaluate how well the recent “frontend compilation” proposal of Pacak and Erdweg [2022] works for Formulog.
- We observe that—unlike for traditional Datalog workloads—the *order* in which logical inferences are made affects Formulog performance by determining the distribution of the SMT workload across threads. This inspires a novel “eager” strategy for parallel Datalog evaluation that eschews the batching of semi-naïve evaluation in favor of eagerly pursuing the consequences of logical derivations with the help of a work-stealing thread pool (Section 4).
- We extend the Formulog interpreter and Soufflé’s code generator to support eager evaluation, leading to arithmetic mean speedups of  $5.2\times$  and  $7.6\times$ , respectively, over using off-the-shelf Soufflé on SMT-heavy Formulog benchmarks; doing so demonstrates that eager evaluation is a practical and effective approach consistent with existing Datalog infrastructure (Section 5).
- We reassess Formulog’s performance in a wider context—showing that Formulog-based analyses can be competitive with previously published, hand-tuned, non-Datalog implementations of a range of SMT-based analyses—and suggest future improvements (Section 6).

## 2 Background

We overview the basics of Datalog (Section 2.1), Soufflé (Section 2.2), and Formulog (Section 2.3). Throughout, we **boldface** a metavariable to indicate zero or more repetitions.

Constructs		Namespaces	
Programs	$\text{prog}^{\text{dl}}$	$::=$	$\text{H}^{\text{dl}}$
Horn clauses	$H^{\text{dl}}$	$::=$	$p(\mathbf{t}) :- A^{\text{dl}}$
Atoms	$A^{\text{dl}}$	$::=$	$p(\mathbf{t}) \mid !p(\mathbf{t}) \mid t = t \mid t \neq t$
Terms	$t$	$::=$	$X \mid n \mid @f^{\text{cpp}}(\mathbf{t})$
Variables	$X$	$\in$	Var
Integers	$n$	$\in$	$\mathbb{Z}$
Predicates	$p$	$\in$	PredVar
C++ functions	$f^{\text{cpp}}$	$\in$	CppVar

Fig. 1. A grammar distilling the standard features of Datalog; we model Soufflé by adding *functor calls*  $@f^{\text{cpp}}(\mathbf{t})$ , which are FFI calls invoking external C++ functions.

## 2.1 Datalog

A Datalog [Ceri et al. 1989; Gallaire and Minker 1978; Green et al. 2013] program is a set of Horn clauses (Figure 1), where each Horn clause is in the form  $p(\mathbf{t}) :- A^{\text{dl}}$ . A Horn clause can be interpreted as a logical implication: the *head* predicate  $p(\mathbf{t})$  holds if all the *body* atoms  $A^{\text{dl}}$  are true. At a high level, Datalog evaluation amounts to making all logical inferences justified by the implications in the program, where each Horn clause is an inference rule in which the body atoms are the premises and the head predicate is the conclusion:

$$p(t_1, \dots, t_m) :- A_1^{\text{dl}}, \dots, A_n^{\text{dl}} \iff \frac{A_1^{\text{dl}} \dots A_n^{\text{dl}}}{p(t_1, \dots, t_m)}$$

An atom  $A^{\text{dl}}$  has one of four forms. In addition to the standard *positive predicates*  $p(\mathbf{t})$ , Datalog typically supports *negative predicates*  $!p(\mathbf{t})$ . A negative predicate holds when it cannot be derived by the program (*negation as failure* [Clark 1977]). *Unification predicates*  $t = t$  unify their two arguments, failing if the unification is impossible. *Inequality predicates*  $t \neq t$  hold when their arguments are not equal. All these predicates are defined over *terms*,  $t$ , which are either variables  $X$  or constants; here we assume, without loss of generality, that each constant is an integer  $n$ . (We discuss Soufflé’s foreign function calls,  $@f^{\text{cpp}}(\mathbf{t})$ , below.)

The predicate symbols in a Datalog program can be split into *strata*, so that the predicate symbol of the head of a rule is not in a lower stratum than any predicate symbol occurring in its body. By splitting a Datalog program into parts and putting a dependency order on them, stratification enables an intuitive and easily computable meaning for negated predicates in programs that meet the requirements of *stratified negation* [Apt et al. 1988; Przymusinski 1988; Van Gelder 1989]. The standard for Datalog, stratified negation sidesteps semantic and computational complications that arise when negation occurs within recursion (cf. the stable model semantics [Gelfond and Lifschitz 1988]), by requiring a predicate symbol  $p$  to be in a strictly higher stratum than predicate symbol  $q$  if there is a rule with predicate  $p(\mathbf{t})$  in the head and negated predicate  $!q(\mathbf{t}')$  in the body.

Datalog evaluation typically proceeds as a sequence of least-fixpoint computations, computing all the tuples in the predicates at the lowest stratum and working upwards, where each stratum is solved using some form of bottom-up evaluation. For example, consider a stratum computing graph transitive closure, in which there is a non-recursive relation `edge` storing the edges of the graph (the input to the stratum) and a recursive relation `reach` defining their transitive closure:

```
reach(X, Y) :- edge(X, Y). // rule trans_edge
reach(X, Z) :- edge(X, Y), reach(Y, Z). // rule trans_step
```

*Naive evaluation* repeatedly runs these inference rules until no new `reach` tuples are derived (i.e., a fixpoint is reached). Running *all* the rules involves significant redundant computation—each iteration re-derives all tuples derived by previous iterations. The de facto standard technique of

*semi-naive evaluation* [Bancilhon 1986] avoids some of this redundancy by executing Horn clauses rewritten to use auxiliary relations indexed by the iteration of the fixpoint computation:

$$\begin{aligned} \Delta(\text{reach})^{[1]}(X, Y) & :- \text{edge}(X, Y). \\ \Delta(\text{reach})^{[i]}(X, Z) & :- \text{edge}(X, Y), \delta(\text{reach})^{[i-1]}(Y, Z). \end{aligned}$$

A predicate symbol  $p^{[i]}$  signifies relation  $p$  at the beginning of step  $i$ , and  $p^{[i+1]} = \Delta(p)^i \cup p^{[i]}$  and  $\delta(p)^{[i]} = \Delta(p)^{[i]} - p^{[i]}$ . Intuitively, the relation  $\delta(p)^{[i]}$  is the set of new tuples derived in iteration  $i$ , and the relation  $\Delta(p)^{[i]}$  is an overapproximation of that set. We have reached a fixpoint for a stratum once all of its  $\delta$ -relations are empty, i.e., we can learn nothing more. Auxiliary relations are used only for predicates that are recursive in the current stratum; other relations are used directly.

Semi-naive evaluation is more complex in the presence of non-linearly recursive rules, where a single rule is rewritten into multiple semi-naive rules referring to different auxiliary relations. Consider the nonlinearly recursive reformulation of rule `trans_step`:

```
reach(X, Z) :- reach(X, Y), reach(Y, Z).           // rule trans_guess
```

The Datalog engine rewrites `trans_guess` into *two* rules, one per occurrence of a recursive predicate:

$$\begin{aligned} \Delta(\text{reach})^{[i]}(X, Z) & :- \delta(\text{reach})^{[i-1]}(X, Y), \text{reach}^{[i]}(Y, Z). \\ \Delta(\text{reach})^{[i]}(X, Z) & :- \text{reach}^{[i]}(X, Y), \delta(\text{reach})^{[i-1]}(Y, Z). \end{aligned}$$

Note that each rule accesses the `reach` relation as it stood at the beginning of that iteration.<sup>2</sup> Readers looking to learn more can consult Datalog surveys [Ceri et al. 1989; Green et al. 2013].

## 2.2 Soufflé

Soufflé [Jordan et al. 2016; Scholz et al. 2016] is a leading Datalog implementation designed specifically for static analysis workloads. It has many features; we focus on those relevant to this paper. Foremost among these, Soufflé is itself a compiler: it compiles Datalog programs to an intermediate representation—instructions on a notional relational algebra machine (RAM)—and then compiles these instructions to C++ code. The resulting code can be linked against external C++ functions, which can be invoked in Soufflé source code as *functor calls*, written  $@f^{\text{CPP}}(t)$  (Figure 1).

A Horn clause is typically compiled into nested for loops. For example, the semi-naive version of the `trans_step` rule (from above) is compiled into these loops (given in pseudo-Python):

```
pfor (x, y) in edge:
    lb = (y, DOMAIN_MIN) # key for lower bound
    ub = (y, DOMAIN_MAX) # key for upper bound
    for (_, z) in  $\delta(\text{reach})^{[i-1]}$ .range_inclusive(lb, ub):
        if (x, z) not in  $\text{reach}^{[i]}$ :
             $\delta(\text{reach})^{[i]}$ .insert((x, z))
```

Soufflé parallelizes the outermost loop using an OpenMP parallel for loop [Dagum and Menon 1998]. This loop iterates over the entire edge relation; the next loop iterates over a slice of the  $\delta(\text{reach})^{[i-1]}$  relation that is extracted by making a range query (here, over the entire domain). Much of the engineering and research behind Soufflé has gone into making these data structure operations efficient. First off, Soufflé uses an optimal indexing strategy minimizing the number of indices that need to be maintained for a relation [Subotić et al. 2018]. Second, Soufflé uses concurrent tries [Jordan et al. 2019a] and B-trees [Jordan et al. 2019b] that have been specialized

<sup>2</sup>Technically, the first rule can have  $\text{reach}^{[i-1]}(Y, Z)$  instead of  $\text{reach}^{[i]}(Y, Z)$ , or alternatively the second rule can have  $\text{reach}^{[i-1]}(X, Y)$  instead of  $\text{reach}^{[i]}(X, Y)$ . This optimization materializes fewer combinations of tuples, since  $\text{reach}^{[i-1]} \subseteq \text{reach}^{[i]}$  (by the monotonicity of Datalog). However, neither Soufflé nor Formulog uses it.

Constructs	
Programs	$\text{prog}^{\text{flg}} ::= \mathbf{T F H}^{\text{flg}}$
Functions	$F ::= \text{fun } f^{\text{flg}}(\mathbf{X}) = e$
Horn clauses	$H^{\text{flg}} ::= p(\mathbf{e}) :- \mathbf{A}^{\text{flg}}$
Atoms	$A^{\text{flg}} ::= p(\mathbf{e}) \mid !p(\mathbf{e}) \mid e == e \mid e != e \mid X \leftarrow e \mid e \rightarrow c(\mathbf{X})$
Constants	$k ::= \text{true} \mid \text{false} \mid n \mid \dots$
Expressions	$e ::= X \mid k \mid c(\mathbf{e}) \mid f^{\text{flg}}(\mathbf{e}) \mid p(\mathbf{e}) \mid \text{let } X = e \text{ in } e \mid$ $\text{if } e \text{ then } e \text{ else } e \mid \text{match } e \text{ with } [c(\mathbf{X}) \rightarrow e]^*$
Additional namespaces	
Type definitions	$T \in \text{TypeVar}$
Function symbols	$f^{\text{flg}} \in \text{FunVar}$
Constructors	$c \in \text{CtorVar}$

Fig. 2. A grammar for core Formulog, which extends Datalog with functional programming via algebraic data type definitions  $T$ , functions  $F$ , and expressions  $e$ . The functional language includes constructors for building terms representing SMT formulas and functions for testing their satisfiability and extracting models.

for Datalog evaluation. These optimizations help make Soufflé a top-performing Datalog engine on static analysis workloads [Fan et al. 2019], as evidenced by its use as the backbone of a slew of static analysis platforms [Antoniadis et al. 2017; Flores-Montoya and Schulte 2020; Grech et al. 2019, 2018; Smaragdakis et al. 2021].

### 2.3 Formulog

Formulog [Bembek et al. 2020b] is a recent Datalog variant for implementing static analyses that use satisfiability-modulo-theories (SMT) solving [de Moura and Bjørner 2011], a key automated reasoning technology that enriches boolean satisfiability (SAT) solving with predicate logic and the ability to reason about common program constructs such as arrays, machine integers, and IEEE floating point numbers.

We focus on a minimal core representation of Formulog (Figure 2). The key characteristic of this representation is that unification predicates  $e = e$  have been replaced with three new types of atoms: check atoms, assignment atoms, and destructor atoms. This representation is simpler to interpret and compile, as it separates out and sequences heterogeneous operations that can occur implicitly within a single unification. A *check atom*  $e == e$  holds if two expressions evaluate to the same value, like an equality predicate. An *assignment atom*  $X \leftarrow e$  assigns the value of an expression  $e$  to a variable  $X$  (which cannot be bound to a value beforehand; atoms are evaluated left-to-right). A *destructor atom*  $e \rightarrow c(\mathbf{X})$  fails if  $e$  does not evaluate to a value with outermost constructor  $c$ ; otherwise, it assigns the constructor’s arguments to the variables  $\mathbf{X}$  (which cannot be bound). We write  $e != e$  to negate a check atom—like an inequality predicate—but assignment and destructor atoms cannot be negated.

Beyond Horn clauses, Formulog supports first-order functional programming and SMT solving.

Formulog embeds a functional language with standard features like algebraic data types and pattern matching. It also provides the ability for functional code to reference Datalog relations by treating a predicate symbol  $p$  as a function symbol. For example, if the tuple  $(\theta, 1)$  is a member of the relation `reach`, the expression `if reach( $\theta, 1$ ) then "yes" else "no"` evaluates to the string "yes", and evaluates to "no" otherwise. Formulog requires the use of predicates-as-functions to be stratified: if evaluating a rule with head predicate symbol  $p$  might run functional code that invokes predicate  $q(\mathbf{e})$ , predicate symbol  $q$  must be in a lower stratum than predicate symbol  $p$ .



Formulog’s SMT support is built on top of its functional programming support: the various SMT term constructors are simply built-in Formulog-level constructors. The built-in function `is_sat` serializes its argument—a complex term representing an SMT formula—into the SMT-LIB standard format [Barrett et al. 2016] and discharges it to an external SMT solver to check for satisfiability. Formulog’s SMT API makes it easy to check validity and extract models, too.

The prototype Formulog system is 20k SLOC Java and uses semi-naive evaluation by default.<sup>3</sup> The Formulog runtime supports parallel evaluation, and each evaluation worker thread interacts with its own stateful, external solver process. Modern SMT solvers support incremental solving [Eén and Sörensson 2003], which benefits from query *locality*: solving is (often) faster if consecutive queries to a given solver instance are similar. Formulog uses a few encoding tricks to increase the number of conjuncts shared between queries on a given thread [Bembenek et al. 2020a].

### 3 Compiling Formulog to Soufflé

This section develops a compiler from Formulog to off-the-shelf Soufflé, first presenting the high-level strategy (Section 3.1) and then implementation details (Section 3.2). In the mean, the generated code has a 2.2× speedup over the Formulog interpreter and uses 38× less memory (Section 3.3).

#### 3.1 Core Translation

The basic compilation strategy is to translate Formulog’s Datalog fragment to Soufflé code and its functional fragment to C++ code accessible to the Soufflé code via external functors (Figure 3).

At the top level (PROG), the translation takes as input a Formulog program consisting of type definitions, function definitions, and Horn clauses, and produces as output (Soufflé) Horn clauses and C++ code. Translating a program consists of four main parts. First, the Formulog type information is used to instantiate a skeleton C++ Formulog runtime. The runtime includes the standard library and interface to external SMT solvers; type information is necessary to complete the representation of terms and their serialization into SMT-LIB. Second, the Horn clauses and function definitions are examined to produce administrative rules that constrain the Soufflé code to respect the semantics of the source Formulog program, such as having the same stratification. Third, the function definitions are compiled to C++ functions using standard compilation techniques for functional languages (the only nonstandard feature being predicates  $p(e)$  invoked as functions; discussed in Section 3.2.3). Finally, each Horn clause in the Formulog program is translated in turn (CLAUSE).

Translating a clause consists of translating the atoms in order, starting with the leftmost body atom and concluding with the head of the rule (CLAUSE). During this process, the compiler keeps track of the variables that are used in the rule, so that it can generate fresh variables on demand. Generic predicates  $p(e)$  (POSPRED) and  $!p(e)$  (rule omitted), equality checks  $e == e$  (CHECK) and  $e != e$  (rule omitted), and assignment  $X \leftarrow e$  (ASSIGN) all compile down to single Soufflé atoms. On the other hand, destructors  $e \rightarrow c(X)$  compile down to *multiple* Soufflé atoms (DESTRUCT), due to a limitation in the current version of Soufflé (Section 3.2.2).

Expressions, which occur as arguments to atoms, are compiled to Soufflé terms two ways. Variables are translated using the identity function (VAR). For all other expressions (NONVAR), the compiler creates a fresh Formulog function definition with that expression as its body and the expression’s free variables as its arguments, and translates the function to C++; the compiler then emits an external functor call invoking the C++ function corresponding to the expression.

#### 3.2 Implementation Details

Our compiler is 4,000 SLOC Java and the runtime is 3,000 SLOC C++.

<sup>3</sup>All SLOC counts given in this paper are of non-blank, non-comment, physical lines.

## Contexts and additional namespaces

Function and code contexts	$\Gamma ::= \cdot \mid \Gamma, f^{\text{flg}} \mapsto f^{\text{cpp}} \mid \Gamma, C$
Logic variable contexts	$\Delta \subseteq \text{Var}$
C++ code	$C \in \text{CppCode}$

## Program translation

$$\boxed{\vdash \llbracket \text{prog}^{\text{flg}} \rrbracket \Rightarrow \mathbf{H}^{\text{dl}}, C}$$

$$\frac{\begin{array}{l} \Gamma_0 = \text{makeFlgRuntime}(\mathbf{T}) \quad \mathbf{H}_{\text{admin}}^{\text{dl}} = \text{makeAdminRules}(\mathbf{F} \mathbf{H}^{\text{flg}}) \\ |\mathbf{F}| = m \quad \forall i \in [0, m). \Gamma_i \vdash \llbracket \mathbf{F}[i] \rrbracket \Rightarrow \Gamma_{i+1} \\ |\mathbf{H}^{\text{flg}}| = n \quad \forall i \in [0, n). \Gamma_{m+i} \vdash \llbracket \mathbf{H}^{\text{flg}}[i] \rrbracket \Rightarrow H_i^{\text{dl}}, \Gamma_{m+i+1} \quad C = \text{extractCode}(\Gamma_{m+n}) \end{array}}{\vdash \llbracket \mathbf{T} \mathbf{F} \mathbf{H}^{\text{flg}} \rrbracket \Rightarrow \mathbf{H}_{\text{admin}}^{\text{dl}} ++ [H_0^{\text{dl}}, \dots, H_{n-1}^{\text{dl}}], C} \text{PROG}$$

(helper functions are defined in the extended version of this paper [Bembenek et al. 2024a])

## Function translation

$$\boxed{\Gamma \vdash \llbracket F \rrbracket \Rightarrow \Gamma}$$

(definition omitted; follows standard techniques)

## Clause translation

$$\boxed{\Gamma \vdash \llbracket H^{\text{flg}} \rrbracket \Rightarrow H^{\text{dl}}, \Gamma}$$

$$\frac{\begin{array}{l} |\mathbf{A}^{\text{flg}}| = n \quad \Delta_0 = \text{vars}(p(\mathbf{e}) :- \mathbf{A}^{\text{flg}}) \\ \forall i \in [0, n). \Gamma_i, \Delta_i \vdash \llbracket \mathbf{A}^{\text{flg}}[i] \rrbracket \Rightarrow \mathbf{A}_i^{\text{dl}}, \Gamma_{i+1}, \Delta_{i+1} \\ \Gamma_n, \Delta_n \vdash \llbracket p(\mathbf{e}) \rrbracket \Rightarrow [p(\mathbf{t}), \Gamma_{\text{final}}, \_ \quad \mathbf{A}_{\text{body}}^{\text{dl}} = \mathbf{A}_0^{\text{dl}} ++ \dots ++ \mathbf{A}_{n-1}^{\text{dl}} \end{array}}{\Gamma_0 \vdash \llbracket p(\mathbf{e}) :- \mathbf{A}^{\text{flg}} \rrbracket \Rightarrow p(\mathbf{t}) :- \mathbf{A}_{\text{body}}^{\text{dl}}, \Gamma_{\text{final}}} \text{CLAUSE}$$

## Atom translation

$$\boxed{\Gamma, \Delta \vdash \llbracket A^{\text{flg}} \rrbracket \Rightarrow \mathbf{A}^{\text{dl}}, \Gamma, \Delta}$$

$$\frac{|\mathbf{e}| = n \quad \forall i \in [0, n). \Gamma_i \vdash \llbracket \mathbf{e}[i] \rrbracket \Rightarrow t_i, \Gamma_{i+1}}{\Gamma_0, \Delta \vdash \llbracket p(\mathbf{e}) \rrbracket \Rightarrow [p(t_0, \dots, t_{n-1})], \Gamma_n, \Delta} \text{POSRED}$$

$$\frac{\Gamma_0 \vdash \llbracket e_1 \rrbracket \Rightarrow t_1, \Gamma_1 \quad \Gamma_1 \vdash \llbracket e_2 \rrbracket \Rightarrow t_2, \Gamma_2}{\Gamma_0, \Delta \vdash \llbracket e_1 == e_2 \rrbracket \Rightarrow [t_1 = t_2], \Gamma_2, \Delta} \text{CHECK} \quad \frac{\Gamma \vdash \llbracket e \rrbracket \Rightarrow t, \Gamma'}{\Gamma, \Delta \vdash \llbracket X \leftarrow e \rrbracket \Rightarrow [X = t], \Gamma', \Delta} \text{ASSIGN}$$

$$\frac{\begin{array}{l} \Gamma \vdash \llbracket e \rrbracket \Rightarrow t, \Gamma' \quad \{X, Y, Z\} \cap \Delta = \emptyset \\ \mathbf{A}_{\text{dctor}}^{\text{dl}} = [X = t, @\text{is\_ctor}_c(X) = Y, \text{move\_barrier}(Y, Z)] \\ \mathbf{A}_{\text{assign}}^{\text{dl}} = [X_0 = @\text{nth}(0, X, Z), \dots, X_{n-1} = @\text{nth}(n-1, X, Z)] \end{array}}{\Gamma, \Delta \vdash \llbracket e \rightarrow c(X_0, \dots, X_{n-1}) \rrbracket \Rightarrow \mathbf{A}_{\text{dctor}}^{\text{dl}} ++ \mathbf{A}_{\text{assign}}^{\text{dl}}, \Gamma', \Delta \cup \{X, Y, Z\}} \text{DESTRUCT}$$

## Expression translation

$$\boxed{\Gamma \vdash \llbracket e \rrbracket \Rightarrow t, \Gamma}$$

$$\frac{}{\Gamma \vdash \llbracket X \rrbracket \Rightarrow X, \Gamma} \text{VAR} \quad \frac{\begin{array}{l} e \notin \text{Var} \quad f^{\text{flg}} \text{ fresh in } \Gamma \quad \mathbf{X} = \text{freeVars}(e) \\ \Gamma \vdash \llbracket \text{fun } f^{\text{flg}}(\mathbf{X}) = e \rrbracket \Rightarrow \Gamma' \quad \Gamma'(f^{\text{flg}}) = f^{\text{cpp}} \end{array}}{\Gamma \vdash \llbracket e \rrbracket \Rightarrow @f^{\text{cpp}}(\mathbf{X}), \Gamma'} \text{NONVAR}$$

Fig. 3. Rules for translating from Formulog to Soufflé. Negative predicates  $!p(\mathbf{e})$  and inequality predicates  $e_1 \neq e_2$  can be translated analogously to their positive counterparts. We use the notation  $++$  for list concatenation.

**3.2.1 Representing Values.** Formulog values are represented as integers in the Soufflé code and as C++ objects in our C++ runtime. The runtime performs hash consing during all value creation, so that there is exactly one C++ object per Formulog value. This allows us to use the memory address of a Formulog object as the integer representation of that value in Soufflé; translating values as they flow between Soufflé and our runtime is as simple as casting.

An alternative would be to embed Formulog values into Soufflé’s type system, which supports algebraic data types. We decided against this approach for four reasons. First, there is not a natural translation between Formulog’s type system and Soufflé’s. For example, Formulog has multiple widths of integers, whereas Soufflé has just one; Formulog data types have ML-style parametric polymorphism, whereas Soufflé has templates. Second, our approach gives us more control over the memoization of terms. Third, it frees us from having to use Soufflé’s internal representation of complex values in the C++ code we generate for Formulog expressions. Fourth, Soufflé (as of v2.4) does not support unpacking complex terms returned from external functors—i.e., getting their arguments—and so most manipulation of terms would need to occur outside Soufflé anyway.

**3.2.2 Compiling Destructors.** Because of a current limitation in Soufflé, each Formulog destructor atom is compiled to *multiple* Soufflé atoms (DESTRUCT; see discussion in the extended version of this paper [Bembenek et al. 2024a]). It should in principle be straightforward to compile a destructor atom to a single Soufflé atom using Soufflé’s support for complex terms; however, this currently leads to an assertion failure within the Soufflé code generator. While we expect this limitation to be fixed in a future version of Soufflé, we do not believe that the current clunky translation leads to a substantial runtime cost.

**3.2.3 Reifying Relations.** In addition to standard functional programming fare, Formulog supports querying relations from functional code, by invoking predicates  $p(e)$  as functions. We need to account for this both in our generated C++ code and Soufflé code. On the C++ side, when the main driver of the executable creates an instance of the Soufflé program (i.e., a value of type `souffle::SouffleProgram`), it stores it into a global variable before executing it. This allows us to generate C++ code that looks up the content of relations while the Soufflé program is running. On the Soufflé side, we create administrative rules to make sure that the stratification of the Soufflé program lines up with the stratification of the Formulog program: the use of predicates as functions must be stratified (as mentioned in Section 2.3), but Soufflé does not see the functional code. For example, a rule defining a nullary relation  $p$  might invoke a function  $f$ , which in turn queries the emptiness of a nullary relation  $q$ . To force Soufflé to put relation  $p$  in a higher stratum than  $q$ , we generate the rule  $p() :- \text{empty}(), !q()$ . (where `empty` is an empty administrative relation).

### 3.3 Evaluation

The code generated by our compiler beats the Formulog interpreter on the case studies and benchmarks from the Formulog paper [Bembenek et al. 2020b] (Figure 4), achieving a mean speedup of 2.2× (min/median/max: 1.0×/1.7×/4.3×) and using 38× less memory in the mean (min/median/max: 6.7×/20×/110×).<sup>4</sup> The baseline interpreter we compared against is an improved version of the one used in the original Formulog paper (e.g., it now completes on all benchmarks, whereas it previously timed out on three); we also manually tuned some of the baseline interpreter’s parameters to achieve better performance, and resolved various performance anomalies.

We ran all experiments on an Ubuntu 22.04 m5.12xlarge Amazon Web Services EC2 instance, with an Intel Xeon processor clocked at 3.1 GHz, 24 physical cores (48 vCPUs), and 192 GiB RAM. We use OpenJDK v17.0.7 as our Java runtime, g++ v11.3.0 as our C++ compiler, Z3 [de Moura and

<sup>4</sup>All means reported in this paper are arithmetic.

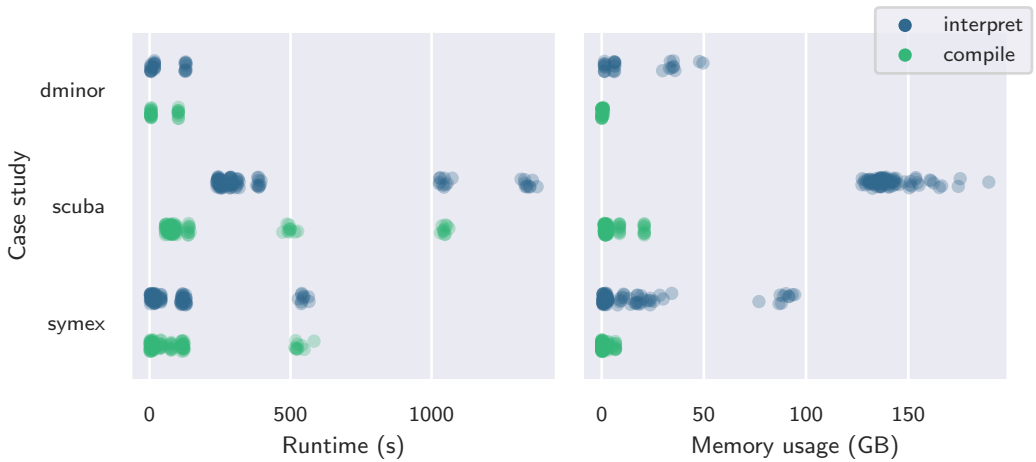


Fig. 4. Formulog compiled to off-the-shelf Soufflé achieves a 2.2× mean speedup over the baseline Formulog interpreter, while also using 38× less memory in the mean. Left is better; we omit compilation time (static analyses are compiled once and executed many times).

[Bjørner 2008] v4.12.2 (commit f7c9c9e) as our SMT solver, and Soufflé v2.4. Both the generated code and the Formulog interpreter are configured to use 40 threads.

To compute the runtime on a benchmark, we take the median of 10 trials. We omit compilation times, on the basis that the Formulog programs in this benchmark suite are static analyses that are intended to be compiled once and then executed many times. Compilation time is dominated by the time it takes to compile the C++ code generated by Soufflé, which can take up to a few minutes.

The benchmarks fall into three case studies representing different SMT-based static analyses, each consisting of 1-1.5K lines of Formulog code (for details, see the original Formulog paper [Bembek et al. 2020b]). We updated the code to work with newer versions of Formulog and to avoid unevaluated expressions in input facts (which the C++ runtime parser does not currently support).

*Refinement Type Checking (dminor)*. This case study is a type checker for Dminor [Bierman et al. 2012], a language combining refinement types and dynamic type tests. We categorize this case study as “SMT heavy” as the SMT solver is frequently invoked during type checking to prove subtyping relations and the termination of expressions (in the longest-running benchmark, 96k external SMT solver calls are made). The SMT formulas are also particularly complex, as they touch on many SMT theories and include constructs like universally quantified formulas. The three benchmarks are synthetic, created by composing all publicly available Dminor programs (that use the core language only) and then copying the composite program  $n$  times, for  $n \in \{1, 10, 100\}$ .<sup>5</sup> For the benchmark with 100 copies of the base program (all-100), there are 4600 input tuples and 1.7 million output tuples; the most computationally intensive part of this case study is the SMT solving. On this case study the compiler achieved a mean speedup of 1.9× (min/median/max: 1.2×/1.2×/3.2×).

*Bottom-Up Points-To Analysis (scuba)*. This context-sensitive, bottom-up points-to analysis for Java uses SMT formulas to summarize methods [Feng et al. 2015]. While SMT solving is used when

<sup>5</sup>The copies are transformed so that they are not syntactically equal, which in turn means that the SMT queries generated by Formulog for each copy are also not syntactically equal. Because the base program exercises many parts of the core Dminor calculus, we believe this approach of Bembek et al. [2020b] to synthesizing benchmarks leads to a reasonable approximation of the workloads that would be induced by variously sized Dminor developments, were they to exist.

function summaries are instantiated, we categorize this case study as “SMT light” because few external SMT calls are made in practice (many potential SMT queries can instead be resolved by basic preprocessing done in the analysis code). Additionally, a relatively small proportion of the analysis (20%) is written in Formulog’s functional fragment; hence, of the three case studies, this one most closely resembles conventional Datalog programs. The 10 benchmarks are substantial Java applications used in the evaluation of the original implementation of this analysis [Feng et al. 2015], drawn primarily from existing Java benchmark suites including DaCapo [Blackburn et al. 2006]. Accordingly, these benchmarks have large numbers of input and output facts, with input fact databases ranging from 65k tuples (weblech) to 1.5 million tuples (xalan), and output fact databases ranging from 4.3 million tuples (polyglot) to 120 million tuples (sunflow). On this case study the compiler achieved a mean speedup of 3.2× (min / median / max: 1.3× / 3.4× / 4.3×).

*Symbolic Execution (symex).* This case study is a bounded symbolic executor for a fragment of LLVM bitcode in the style of KLEE [Cadaru et al. 2008]. The 10 benchmark programs are compiled C programs that perform operations like array sorting, solving logic puzzles, and interpreting bitcode. As the benchmarks are synthetic and need to fit within the supported fragment of LLVM, the source code for each benchmark is generally small in terms of lines of code; however, because many of the programs loop over symbolic data, the number of states to symbolically explore can be quite large. We categorize this case study as “SMT heavy” as SMT calls are made frequently, any time the symbolic executor reaches a branch point conditioned on symbolic data (220k external SMT solver calls are made in sort-7, the benchmark with the most SMT calls). The largest input fact database is 6000 tuples (prioqueue-6), but the largest output fact database is 28 million tuples (sort-7). On this case study the compiler achieved a mean speedup of 1.4× (min / median / max: 1.0× / 1.4× / 2.0×).

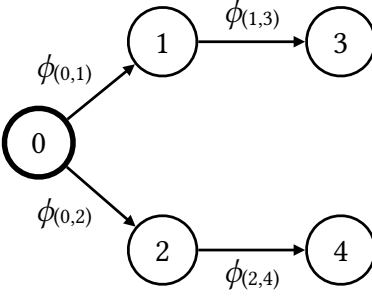
#### 4 Eager Evaluation

The Soufflé code generated by our compiler outperforms the Formulog interpreter when both perform semi-naïve evaluation. This is hardly surprising, as using Soufflé has many advantages over using the Formulog prototype, such as compilation instead of interpretation, C++ instead of Java, and data structures specialized for Datalog instead of generic ones.

What is surprising is that we can do better, and with relatively little effort. There are many possible ways to evaluate a Datalog program, which is declarative: a Datalog program comprises some logical inference rules with no stipulations as to how to discover the rules’ consequences. We need not be beholden to the standard strategy. In fact, one might even expect that new Datalog variants would put different stresses on Datalog engines than traditional Datalog workloads, and thus would benefit from alternative Datalog evaluation strategies. (In fact, even different *Datalog* workloads benefit from different implementation techniques [Fan et al. 2019].)

Formulog indeed differs from traditional Datalog: external SMT solving is typically one of the most expensive parts of evaluating a Formulog program. This section presents *eager evaluation*, an alternative, worklist-based strategy for parallel Datalog evaluation that achieves a quasi-depth-first search of the logical inference space by submitting inference tasks to a work-stealing thread pool.<sup>6</sup> Compared to semi-naïve evaluation, which performs a breadth-first search, eager evaluation entails a different distribution of SMT calls across worker threads (and thus external SMT solvers); in practice, it tends to lead to better SMT solving times. For example, on some Formulog programs, eager evaluation ends up putting similar SMT calls on the same worker thread, leading to more

<sup>6</sup>In a work-stealing thread pool [Arora et al. 1998; Blumofe and Leiserson 1999; Mohr et al. 1990], each worker thread maintains a last-in-first-out (LIFO) stack of work items. Since work items in our setting correspond to logical derivations, the LIFO discipline leads to the most recent derivations being explored first; see additional discussion in Section 4.2.



```

(* base case *)
reach(0, `true`).

(* recursive case *)
reach(Y, `Phi /\ Psi`) :-
  reach(X, Phi),
  edge(X, Y, Psi),
  is_sat(`Phi /\ Psi`) = true.
  
```

Fig. 5. A Formlog program computes reachability over a tree labeled with SMT propositions  $\phi_{(i,j)}$  by using the built-in constructor for conjunction ( $\wedge$ ) and the built-in function `is_sat` to check satisfiability. Backticks demarcate SMT formulas.

effective incremental SMT solving. Using eager evaluation, the Formlog interpreter beats the semi-naive code generated by Soufflé on SMT-heavy benchmarks (Section 5.3).

This section motivates eager evaluation (Section 4.1), describes the high-level algorithm (Section 4.2), and proves its correctness (Section 4.3).

#### 4.1 Motivating Eager Evaluation

To motivate eager evaluation, consider the task of computing reachability on trees where the edges are labeled with logical propositions. The root node is trivially reachable. Some other node  $n$  is reachable if its parent  $m$  is reachable from the root node via a  $k$ -hop path with edge labels  $\phi_1, \dots, \phi_k$ , there is an outgoing edge from node  $m$  with label  $\psi$  to a child node  $n$ , and the conjunction  $\phi_1 \wedge \dots \wedge \phi_k \wedge \psi$  is satisfiable—that is, reachability is as in a directed graph, modulo satisfiability of path conditions. This setting exemplifies a fundamental problem in real static analyses, mimicking how symbolic execution explores program execution trees.

For the sake of concreteness, say we run the Formlog program computing this type of graph reachability on a sample tree (Figure 5), using a single thread of execution. If we use semi-naive evaluation, the program will discover nodes in a breadth-first order: 0-1-2-3-4. Doing so involves four SMT calls, which occur in this order:  $\phi_{(0,1)}$ ,  $\phi_{(0,2)}$ ,  $\phi_{(0,1)} \wedge \phi_{(1,3)}$ , and  $\phi_{(0,2)} \wedge \phi_{(2,4)}$ . Because adjacent calls do not share any conjuncts, the SMT solver is not able to naturally perform incremental SMT solving; in the worst case, it would have to clear its state between calls that *do* share conjuncts. And, while Formlog does use some tricks to encode conjuncts in the SMT solver’s state so that they can be disabled and enabled in subsequent calls [Bembenek et al. 2020a], these techniques are necessarily limited and introduce overhead.

In contrast, if one were to compute reachability over the example graph using a Datalog evaluation that simulates depth-first search, we might encounter the nodes in the order 0-1-3-2-4, inducing SMT calls in the order  $\phi_{(0,1)}$ ,  $\phi_{(0,1)} \wedge \phi_{(1,3)}$ ,  $\phi_{(0,2)}$ , and  $\phi_{(0,2)} \wedge \phi_{(2,4)}$ . This order naturally leads to incremental SMT solving, since the first two calls share a conjunct, as well as the last two calls. For example, by checking the satisfiability of the proposition  $\phi_{(0,1)}$ , the solver might learn information useful for checking the satisfiability of the conjunction  $\phi_{(0,1)} \wedge \phi_{(1,3)}$ .

In general, semi-naive evaluation computes a breadth-first search through the logical inference space: it computes derivations of height  $k$  only after it has computed all derivations of height less than  $k$ . For the example of graph reachability, this quite literally matches breadth-first graph traversal. If we were in the world of Datalog and concerned solely with computing structural reachability in the graph (irrespective of the satisfiability of path conditions), there would not be

an obvious cost to performing breadth-first search compared to some other traversal. However, given that we are in the world of Formulog and need to respect the satisfiability of path conditions, order *does* matter: depth-first search leads to more opportunities for incremental SMT solving. Put another way, we want a Datalog evaluation strategy that prioritizes computing the consequences of the most recently derived fact—i.e., finding those facts where the most recently derived fact appears in the proof tree—instead of processing facts in order of the height of their proof trees.

This is what motivates eager evaluation: a Datalog evaluation strategy that, by being closer to depth-first search, leads to different and hopefully advantageous SMT workloads for Formulog programs, while staying amenable to effective parallelization.

#### 4.2 Eager Evaluation: Parallel, Most-Recent-First Search

Eager evaluation explores the inference space using a parallelized, DFS-like search. Making derivations in a strict depth-first order would lead to poor parallelism; instead, eager evaluation performs a more relaxed traversal strategy we think of as “most-recent-first search.” This retains the spirit of DFS—it prioritizes processing the most recently derived facts—while being naturally parallelizable.

Similar to semi-naive evaluation, eager evaluation is applied one stratum at time, making it compatible with stratified negation [Apt et al. 1988; Przymusiński 1988; Van Gelder 1989] and stratified aggregation [Mumick et al. 1990]. It is a worklist-based scheme: initial work items are submitted to the worklist; work items recursively submit new work items; evaluation is finished once the worklist is empty. To both parallelize evaluation and achieve a most-recent-first order, eager evaluation uses a work-stealing thread pool for the worklist [Arora et al. 1998; Blumofe and Leiserson 1999; Mohr et al. 1990]. Each worker thread in the pool maintains a dequeue of work items that it pops from the back using the last-in-first-out stack discipline; since a work item corresponds to a new inference to explore, this discipline prioritizes recent derivations, and thus approximates a depth-first traversal through the logical inference space. If a worker thread’s dequeue becomes empty, it steals from the front of a random thread’s dequeue using the first-in-first-out queue discipline, leading to lower thread contention, and often good locality.

Each work item submitted to the thread pool represents a rule to evaluate. There are two types of work items, representing different types of rules. The first type represents a rule that is not recursive in the current stratum; that is, all the body predicates are defined in previous strata. These are the initial work items (one per non-recursive rule). The second type of work item represents a recursive rule that has been specialized to a particular fact  $p(\mathbf{n})$ . To specialize a rule, 1) unify a body atom  $p(\mathbf{t})$  with the fact  $p(\mathbf{n})$  to get a substitution from variables to constants; 2) apply this substitution across the rule; and 3) reorder the rule so that the body atom  $p(\mathbf{n})$  is first. For example, consider the `trans_step` rule for graph transitive closure (Section 2.1):

$$\text{reach}(X, Z) \text{ :- edge}(X, Y), \text{ reach}(Y, Z).$$

Specialized to the fact  $\text{reach}(\theta, 1)$ , the rule would become

$$\text{reach}(X, 1) \text{ :- reach}(\theta, 1), \text{ edge}(X, \theta).$$

Specialization can fail if the body atom  $p(\mathbf{t})$  cannot be unified with the fact  $p(\mathbf{n})$ ; for example, the body atom is  $p(\theta)$  and the fact is  $p(1)$ .

When a work item is processed, it computes the direct consequences of the rule it corresponds to, with respect to the *current contents* of relations. This differs from semi-naive evaluation, which refers to the contents of relations at the beginning of the iteration (i.e., it indexes into relations of the form  $p^{[i]}$ ); eager evaluation does not have explicit iterations and instead works in an unbatched, “tuple at a time” fashion. The fact that rule evaluation uses more up-to-date relations than in

**Algorithm 1** Function `WORKERTHREAD(i)` gives the core logic performed by each worker thread  $i$  in eager evaluation. For each worker thread  $i$ , there is a worklist (deque)  $D_i$ ; for each predicate  $p$ , there is a fact database  $DB_p$ . Worklists and databases must be thread safe.

---

```

1: function WORKERTHREAD( $i$ )
2:   loop
3:      $\triangleright$  Dequeue a work item (a rule to evaluate)  $\triangleleft$ 
4:      $R \leftarrow D_i.dequeueBack()$   $\triangleright$  Use LIFO discipline for local worklist
5:     while  $R = \perp \wedge workLeft()$  do  $\triangleright$  Scan other threads if local worklist is empty
6:        $j \leftarrow chooseRandomThread()$ 
7:        $R \leftarrow D_j.dequeueFront()$   $\triangleright$  Use FIFO discipline if stealing from another thread
8:     if  $R = \perp$  then  $\triangleright$  No work left in thread pool
9:       break
10:     $\triangleright$  Evaluate rule  $R$ , eagerly yielding facts as they are derived (i.e., no batching)  $\triangleleft$ 
11:    for all  $p(\mathbf{n}) \in EVALRULE(R)$  do
12:      if  $DB_p.addIfAbsent(\mathbf{n})$  then  $\triangleright$  True when tuple  $\mathbf{n}$  is novel
13:        for all  $R' \in specializeRules(p(\mathbf{n}))$  do
14:           $D_i.enqueueBack(R')$   $\triangleright$  Insert work item in back of current thread's worklist

15: function EVALRULE( $p_0(\mathbf{t}_0) :- p_1(\mathbf{t}_1), p_2(\mathbf{t}_2), \dots, p_n(\mathbf{t}_n).$ )
16:    $\sigma_1 = \lambda x. \perp$   $\triangleright$  Start with empty substitution
17:   for all  $\mathbf{n}_1 \in DB_{p_1}.query(\sigma_1(\mathbf{t}_1))$  do  $\triangleright$  Iterate through tuples matching key
18:      $\sigma_2 = \sigma_1 \circ [\mathbf{t}_1 \mapsto \mathbf{n}_1]$   $\triangleright$  Compose substitutions
19:     for all  $\mathbf{n}_2 \in DB_{p_2}.query(\sigma_2(\mathbf{t}_2))$  do
20:        $\sigma_3 = \sigma_2 \circ [\mathbf{t}_2 \mapsto \mathbf{n}_2]$ 
21:       ...
22:       for all  $\mathbf{n}_n \in DB_{p_n}.query(\sigma_n(\mathbf{t}_n))$  do
23:          $\sigma_{n+1} = \sigma_n \circ [\mathbf{t}_n \mapsto \mathbf{n}_n]$ 
24:         yield  $p_0(\sigma_{n+1}(\mathbf{t}_0))$   $\triangleright$  Eagerly yield derived facts (do not batch them)

```

---

semi-naive evaluation does not impact correctness, but it does mean that eager evaluation can materialize some redundant combinations of tuples avoided in semi-naive evaluation.

If a new fact  $p(\mathbf{n})$  is derived for some constants  $\mathbf{n}$ , we add the fact to the  $p$  relation and immediately use it to create new work items. Say there are  $k$  occurrences of predicates shaped like  $p(\mathbf{t})$  in the bodies of rules that are recursive in the current stratum; the algorithm will try to submit  $k$  new work items, one for each occurrence, specializing the body atom  $p(\mathbf{t})$  in the relevant rule to the new fact  $p(\mathbf{n})$ . Fewer than  $k$  work items might be submitted if specialization fails in some case.

**4.2.1 Pseudocode.** Algorithm 1 gives pseudocode for the logic implemented by each worker thread  $i$  during eager evaluation. The function `WORKERTHREAD` gives the core procedure performed by thread  $i$ , which loops until there is no work left to do (line 8). In each iteration, thread  $i$  first tries to dequeue a work item from the back of its own deque (line 4); if this fails because the deque is empty (the returned work item is  $\perp$ ), it repeatedly tries to steal a work item from the deques of the other threads (lines 5-7), scanning until it successfully steals a work item or the function call `workLeft()` returns false (indicating that all threads are trying to steal work items, and so all deques are empty). In practice, the work-stealing thread pool—not code implemented by the Datalog system engineer—distributes work items and decides when to terminate the threads. Once thread  $i$  has a work item in the form of a rule  $R$ , the thread evaluates rule  $R$  using the function



`EVALRULE`; the evaluation code (lines 16-24) is standard for Datalog, except that facts are eagerly yielded as they are derived (line 24). That is, the function `EVALRULE` returns a fact *generator*, instead of a concrete set of facts. For each generated fact, thread  $i$  checks if that fact is novel (line 12); if so, it adds a work item to the end of its deque for each rule specialized to that fact (lines 13-14).

Because the code in the `EVALRULE` function is very similar to the code used to evaluate a rule in semi-naive evaluation (cf. Section 2.1) and uses the same set of data structure operations (e.g., looping over the index of a relation), we are able to heavily reuse existing Datalog infrastructure when implementing eager evaluation, as we demonstrate in Section 5.

### 4.3 Correctness of Eager Evaluation

Eager evaluation is correct; in particular, one can prove the following theorem (see the extended version of this paper [Bembenek et al. 2024a] for the complete proofs of all theorems):

**THEOREM 4.1 (CORRECTNESS).** *Given an arbitrary Datalog program, eager evaluation and semi-naive evaluation derive exactly the same facts.*

The proof of Theorem 4.1 rests on two lemmas, both of which make a claim about the behavior of eager evaluation on a given stratum (as opposed to the complete Datalog program). First, eager evaluation is sound on the stratum: it derives *only* the facts derived by semi-naive evaluation (Lemma 4.2). Second, eager evaluation is complete on the stratum: it derives *all* the facts derived by semi-naive evaluation (Lemma 4.3).

**LEMMA 4.2 (SOUNDNESS).** *Fixing the facts derived in previous strata, if eager evaluation derives a fact  $a_0$  in the current stratum, semi-naive evaluation also derives fact  $a_0$  in the current stratum.*

**LEMMA 4.3 (COMPLETENESS).** *Fixing the facts derived in previous strata, if semi-naive evaluation derives a fact  $a_0$  in the current stratum, eager evaluation also derives fact  $a_0$  in the current stratum.*

Completeness is significantly more complex to prove than soundness; in fact, it relies on some (mild) assumptions about the behavior of the concurrent mechanisms used to implement Algorithm 1.

Completeness is challenging to prove because eager evaluation is a concurrent algorithm that uses little explicit synchronization, and so there appears to be the possibility that data races result in the failure to derive a fact. For example, consider evaluating the rule  $s(3) :- p(1), q(2)$ . Call this rule  $R$ , and assume that the predicates  $p$  and  $q$  are recursive in the current stratum. In a multithreaded setting, the derivation of the facts  $p(1)$  and  $q(2)$  might lead to two work items that are processed concurrently, one specialized to fact  $p(1)$  and one specialized to fact  $q(2)$ . Because of data races between the writing and reading of relations, one might worry that *both* work items fail to derive the fact  $s(3)$ —i.e., could it be that the derivation of the fact  $q(2)$  is not visible in the work item specialized to  $p(1)$  and vice versa? Thankfully, no! If one work item fails because of a data race, the other one necessarily succeeds. Here, we informally demonstrate why eager evaluation is complete on this example; see the extended version of this paper [Bembenek et al. 2024a] for a formal proof of the general case.

When facts  $p(1)$  and  $q(2)$  are derived, these events occur (line numbers refer to Algorithm 1):

- Write  $w_{1,p}$  (line 12): the fact  $p(1)$  is added to data structure  $DB_p$ . Subsequently, Work Item 1 specializing rule  $R$  on  $p(1)$  is submitted to the thread pool (line 14).
- Execution start  $x_{1,p}$  (line 4 or 7): Work Item 1 is dequeued to be executed.
- Read  $r_{1,q}$  (line 19): the data structure  $DB_q$  is read when Work Item 1 is processed.
- Write  $w_{2,q}$  (line 12): the fact  $q(2)$  is added to data structure  $DB_q$ . Subsequently, Work Item 2 specializing rule  $R$  on  $q(2)$  is submitted to the thread pool (line 14).
- Execution start  $x_{2,q}$  (line 4 or 7): Work Item 2 is dequeued to be executed.

- Read  $r_{2,p}$  (line 19): the data structure  $DB_p$  is read when Work Item 2 is processed.

The completeness of eager evaluation rests on a transitive happens-before relationship  $\rightarrow$  between these events; we require the happens-before relationship to meet the following conditions:

- (1) Reads and writes on the same data structure are ordered:

$$(r_{1,q} \rightarrow w_{2,q} \vee w_{2,q} \rightarrow r_{1,q}) \wedge (r_{2,p} \rightarrow w_{1,p} \vee w_{1,p} \rightarrow r_{2,p}).$$

This condition would be satisfied by using linearizable concurrent data structures [Herlihy and Wing 1990], a common notion of concurrent data structure correctness.

- (2) Reads that occur in a work item happen after that work item starts executing:  $x_{1,p} \rightarrow r_{1,q} \wedge x_{2,q} \rightarrow r_{2,p}$ . In general, the happens-before relation should respect program order.
- (3) Writes happen before the work items they spawn:  $w_{1,p} \rightarrow x_{1,p} \wedge w_{2,q} \rightarrow x_{2,q}$ . This condition is a basic visibility requirement. In the pseudocode, it would be sufficient if each deque  $D_i$  is linearizable; in practice, a thread pool would likely guarantee this condition by performing some form of synchronization when it handles the distribution of work items.

These mild assumptions are reasonable to meet in practice.

To show that fact s(2) is derived, it is enough to show that  $w_{1,p} \rightarrow r_{2,p}$  holds or  $w_{2,q} \rightarrow r_{1,q}$  holds—that is, the write of p(1) is visible in Work Item 2 or the write of q(2) is visible in Work Item 1. By Assumption (1), we have the disjunction  $r_{1,q} \rightarrow w_{2,q} \vee w_{2,q} \rightarrow r_{1,q}$ . If the second disjunct is true, Work Item 1 will derive fact s(2). Otherwise, we have (read left-to-right, top-to-bottom):

$$\begin{array}{llll} w_{1,p} \rightarrow x_{1,p} & \text{(Assumption (3))} & x_{1,p} \rightarrow r_{1,q} & \text{(Assumption (2))} & r_{1,q} \rightarrow w_{2,q} & \text{(Disjunct case)} \\ w_{2,q} \rightarrow x_{2,q} & \text{(Assumption (3))} & x_{2,q} \rightarrow r_{2,p} & \text{(Assumption (2))} & & \end{array}$$

Thus, by transitivity, we have that  $w_{1,p} \rightarrow r_{2,p}$  holds, and Work Item 2 will derive fact s(2).

## 5 Eager Evaluation in Practice

This section discusses our experience adding eager evaluation to the Formulog interpreter (Section 5.1) and Soufflé’s code generator (Section 5.2), and shows that eager evaluation is a practical and effective strategy for evaluating Formulog programs (Section 5.3): the eager evaluation mode of the *interpreter* achieves a mean speedup of 5.2× over the semi-naïve code generated by Soufflé on SMT-heavy benchmarks, and the eager evaluation extension to (compiled) Soufflé achieves a mean 1.8× speedup over the Formulog interpreter’s eager evaluation mode and a mean 7.6× speedup over off-the-shelf Soufflé. Thus, Formulog is an example of a Datalog variant that benefits from a non-standard evaluation technique; moreover, this technique can be built on top of existing Datalog infrastructure with relatively low effort.

### 5.1 Extending the Formulog Interpreter

Our implementation of eager evaluation in the Formulog interpreter consists of two parts: a short method that reorders the bodies of semi-naïve rules so that  $\delta$ -predicates come first,<sup>7</sup> and a class that evaluates a stratum of the Formulog program using eager evaluation (200 lines of Java). The classes for eager evaluation and semi-naïve evaluation both descend from a shared abstract class (200 lines); both use a work-stealing thread pool (a `java.util.concurrent.ForkJoinPool` instance) to handle parallelism; and both store relations in the same indexed data structures built around `java.util.concurrent.ConcurrentSkipListSet` instances. The remaining logic (and code) in each class is also very similar, with the main difference occurring when a new tuple is derived: in

<sup>7</sup>Our implementations *lazily* specialize rules to new facts—i.e., during work item evaluation, not during work item creation (as in Algorithm 1). Recall that  $\delta$ -relations are auxiliary relations used to focus traditional semi-naïve evaluation on newly derived tuples (Section 2.1). When a work item is processed, the given rule is run on a singleton, mock  $\delta$ -relation containing just the tuple to specialize on. Since the mock  $\delta$ -predicate is the first atom, this has the effect of specializing the rule.

eager evaluation, new work items are submitted corresponding to rules specialized on that tuple; in semi-naive evaluation, the new tuple is added to an auxiliary relation.

## 5.2 Extending Soufflé’s Code Generator

Our extension to Soufflé’s code generator consists of three parts: a short class for reordering semi-naive rule bodies so that  $\delta$ -predicates are first, modifications to the code generator for semi-naive rules, and modifications to the code generator for the data structures representing relations. The latter two occur during the translation from relational algebra machine (RAM) instructions to C++. RAM instructions capture high-level imperative operations, like scanning the index of a data structure, or inserting a tuple into a relation. To support eager evaluation, we change the translation of some RAM instructions. For example, we change insert instructions to insert a new tuple into the base relation (instead of an auxiliary relation), and to submit new work items representing rules specialized on that tuple. We also modify code generation so that the logic for evaluating a rule is wrapped up in its own function; consequently, a work item’s single operation is to invoke a “rule function” on the new tuple. Work items are submitted to a `task_group` instance from Intel’s oneAPI Thread Building Blocks (oneTBB) concurrency library<sup>8</sup>; this uses a work-stealing thread pool on the backend to process tasks.

In addition to modifying code generation for rule evaluation, we modify code generation for the data structures representing relations. This is required because of a limitation in Soufflé’s concurrent data structures, which do not support concurrent reads and writes. Because there are distinct reading and writing phases in semi-naive evaluation, the Soufflé implementors have added only the synchronization necessary for multiple reads happening simultaneously or multiple writes happening simultaneously. To get around this, we use instances of oneTBB’s `concurrent_set` class for indices where necessary. These sets are based on concurrent skip lists and are much slower than Soufflé’s specialized B-trees and tries. Accordingly, we use oneTBB’s concurrent sets for an index only if it is accessed when evaluating a non-“delta” recursive predicate in a rule body (a necessary condition for an index being read and written concurrently). Programs with only linearly recursive rules can thus get away with using just Soufflé’s data structures.

All in all, our modifications required adding just 500 lines of C++ to Soufflé (which is 70k lines of C++, including code not involved in compilation). The eager evaluation code generator can in principle be used to compile arbitrary Datalog programs, and is agnostic to the fact that we use it as part of a compiler for Formulog programs. However, our extension does not currently support all of Soufflé’s RAM instructions (we focus on those used during our case studies), nor all of Soufflé’s additional machinery (e.g., provenance tracking). To link with code generated for eager evaluation, we made minor modifications to the Formulog C++ runtime (for example, using oneTBB’s abstraction for thread-local variables instead of OpenMP’s).

## 5.3 Evaluation

We first summarize the experimental results across the benchmark suite (Section 5.3.1), and then detail the results for each case study (Sections 5.3.2 and 5.3.3). We also evaluate how well eager evaluation scales with the number of threads compared to semi-naive evaluation (Section 5.3.4).

**5.3.1 Summary.** Eager evaluation leads to speedups on SMT-heavy benchmarks (Figure 6). In particular, on the `dminor` refinement type checker and the `symex` symbolic executor, the interpreter’s eager evaluation mode (“interpret-eager”) achieves a 5.2× mean speedup (min/median/max: 0.41×/1.2×/31×) over the code generated by our initial compiler (“compile”); in turn, using the eager evaluation extension to Soufflé (“compile-eager”) achieves a 1.8× mean speedup (min/median/

<sup>8</sup><https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html>

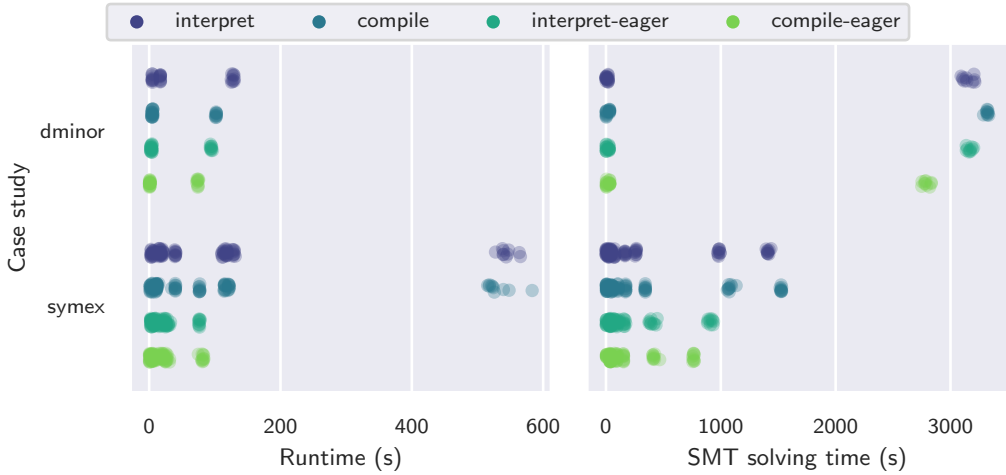


Fig. 6. Eager evaluation achieves speedups on SMT heavy benchmarks over semi-naive evaluation, primarily by improving SMT solving times (eager evaluation’s finer-grained parallelism also helps to a small degree). Note that the *CPU time* of SMT solving (as shown in the right plot) does not capture the full extent to which eager evaluation’s redistribution of SMT work can lead to lower SMT solving *wall-clock time* (e.g., eager evaluation might induce two SMT queries to be solved in parallel instead of sequentially; the CPU time spent in SMT solving would be the same, but the wall-clock time would be less).

max:  $0.93\times/1.3\times/4.1\times$ ) over the interpret-eager mode. This section omits results for the scuba points-to analysis case study, where eager evaluation is not remotely competitive with our initial compiler to off-the-shelf Soufflé. The scuba case study uses little SMT solving and is most similar to traditional Datalog workloads; additionally, eager evaluation reorders rules so that  $\delta$ -predicates come first, which happens to be a very inefficient join ordering strategy here.

Eager evaluation uses more memory than semi-naive evaluation: interpret-eager uses mean  $1.6\times$  as much memory as the baseline interpreter (“interpret”) (min/median/max:  $0.96\times/1.1\times/5.2\times$ ); compile-eager uses mean  $1.1\times$  as much memory as compile (min/median/max:  $0.92\times/1.1\times/1.5\times$ ).

Eager evaluation’s approach to parallelism is finer-grained than the parallel for loops of typical implementations of semi-naive evaluation, and in general the eager evaluation modes demonstrate higher CPU utilization in our experiments (albeit with some variability). Interpret-eager uses mean  $3.9\times$  as many CPUs as interpret (min/median/max:  $0.93\times/1.9\times/12\times$ ); compile-eager uses mean  $1.6\times$  as many CPUs as compile (min/median/max:  $0.19\times/1.1\times/7.1\times$ ).

We also measured the amount of “work” performed by each strategy. We define this quantity as the number of tuple accesses made during the evaluation of a rule; if evaluation takes the form of a nested for loop where loops iterate over relation contents, it is the total number of iterations across all loops. Given a fixed program and input, semi-naive evaluation will always perform the same amount of work (which will, in general, contain some redundant combinations of tuples). Not so for eager evaluation, where the outcome of data races can lead to a different amount of work being performed (which might happen to be more or less than the work done by semi-naive evaluation). In our experiments, interpret and interpret-eager do almost exactly the same amount of work, but compile does slightly more work ( $1.1$ - $1.2\times$ ) than compile-eager.<sup>9</sup>

<sup>9</sup>For the dminor and symex case studies, interpret and compile modes both use a  $\delta$ -predicate-first heuristic for reordering atoms in rule bodies, as that ordering is more efficient on these benchmarks than the default left-to-right order. Since this is

**5.3.2 Refinement Type Checking (*dminor*).** Interpret-eager has a mean 1.3× speedup (min/median/max: 1.1×/1.3×/1.7×) over compile on the *dminor* case study, while compile-eager has a mean 2.8× speedup (min/median/max: 1.3×/3.0×/4.1×) over interpret-eager.

Eager evaluation is not universally more effective than compile at grouping similar SMT calls on the same thread: compile has mean 1.2× (min/median/max: 0.97×/1.2×/1.3×) as many SMT cache misses as interpret-eager, but 0.94× (min/median/max: 0.80×/1.0×/1.0×) as many as compile-eager.<sup>10</sup> This suggests that SMT calls in this case study do not follow the pattern identified in Section 4. Despite this, eager evaluation can still lead to better SMT solving CPU times than compile: SMT solving can be up to 1.2× faster with both interpret-eager and compile-eager. Thus, eager evaluation leads to an advantageous distribution of the SMT workload, despite taking less advantage of SMT caching. This perhaps reflects the difficulty of predicting SMT solver performance, which can vary widely with small perturbations to solver state and query framing (e.g., the choice of SMT variable names can affect how long solving takes and whether the result is sat or unknown<sup>11</sup>).

**5.3.3 Symbolic Execution (*symex*).** Interpret-eager achieves a mean 6.3× speedup (min/median/max: 0.4×/1.1×/31×) over compile on the *symex* benchmarks, while compile-eager manages a 1.5× speedup (min/median/max: 0.9×/1.2×/3.5×) over interpret-eager.

The SMT story is more straightforward for this case study: eager evaluation more effectively groups together similar SMT calls on the same thread, and this leads to faster SMT solving times. In particular, compile has mean 2.5× (min/median/max: 0.34×/1.4×/5.7×) as many SMT cache misses as interpret-eager and mean 5.7× (min/median/max: 0.3×/3.6×/18×) as many as compile-eager. In terms of CPU time spent SMT solving, this gives a mean speedup of 1.4× over compile for both interpret-eager (min/median/max: 0.28×/1.5×/2.7×) and compile-eager (min/median/max: 0.24×/1.6×/2.8×). Thus, as anticipated in Section 4, symbolic execution discharges SMT queries in a pattern well matched to eager evaluation.

**5.3.4 Scaling.** We also tested how well eager evaluation and semi-naive evaluation scale as the number of threads is varied. We ran the compile and compile-eager modes on six benchmarks: the longest-running *dminor* benchmark (all-100), and the longest-running configuration of each of the five template *symex* benchmarks (interp-6, numbrix-sat, prioqueue-6, shuffle-5, and sort-7).<sup>12</sup> We varied the number of threads between 1, 8, 16, 24, 32, 40, and 48, with five trials per configuration. These experiments were run on an Ubuntu 22.04 c5.12xlarge Amazon Web Services EC2 with an Intel Xeon processor clocked at 3.6 GHz, 24 physical cores (48 vCPUs), and 96 GiB RAM.

At the baseline of using a single thread, compile-eager outperforms compile by a mean speedup of 1.7× (min/median/max: 0.9×/1.2×/4.3×). This speedup is slightly less than the mean 2.1× speedup compile-eager has over compile in terms of SMT solving times (min/median/max: 0.9×/1.5×/5.2×), an indication of the overhead of eager evaluation relative to semi-naive evaluation.

On most benchmarks, compile-eager scales better than semi-naive evaluation, achieving higher speedups over single-threaded evaluation as the number of threads is varied (Figure 7). Compile-eager is typically able to more fully take advantage of additional threads, achieving higher speedups than compile for a given thread count, and continuing to get performance improvements past

---

consistent with the atom ordering used by eager evaluation, comparing the amount of work done by semi-naive and eager modes is apples-to-apples (in general, the amount of work done in a rule depends on the order of body atoms).

<sup>10</sup>The Formulog runtime uses encoding tricks to maintain a per-solver cache of conjuncts [Bembenek et al. 2020a]; we count an SMT cache miss for each relevant conjunct that is not in the cache when an SMT call is made.

<sup>11</sup>See <https://github.com/Z3Prover/z3/issues/909/> and <https://github.com/Z3Prover/z3/issues/4600>.

<sup>12</sup>We omit the interpreted Formulog modes on the basis that they are less optimized than the compiled modes, and so how well they scale is a less interesting question.

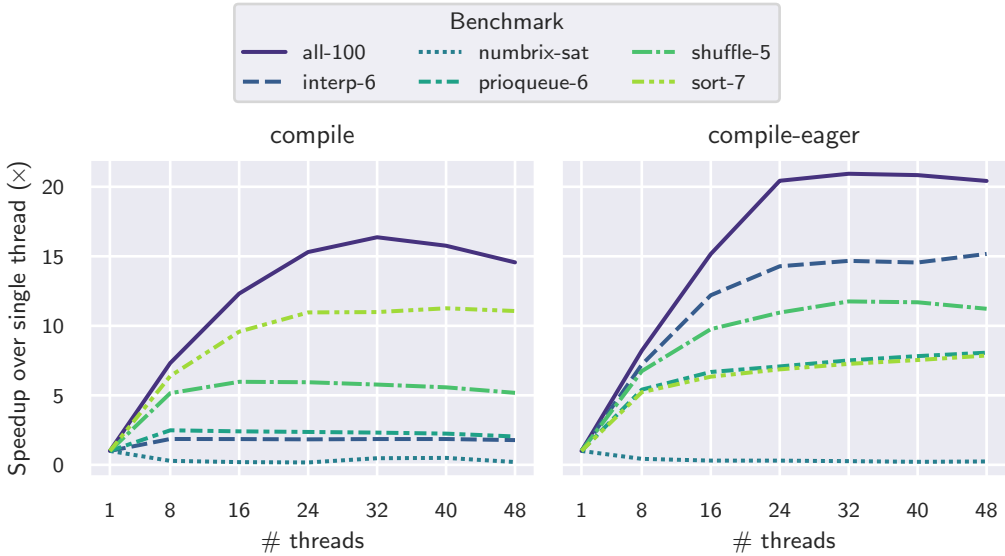


Fig. 7. Eager evaluation typically takes better advantage of additional threads than semi-naive evaluation, achieving higher speedups (relative to single-threaded evaluation) for a given thread count, and continuing to improve performance past the thread count where semi-naive evaluation’s performance stalls or decreases.

the thread count where compile’s performance begins to stall or decrease. Nonetheless, most of compile-eager’s performance is achieved by 32 threads, with diminishing returns beyond that point.

Two benchmarks are outliers to the general trend. First, single-threaded evaluation is best for both evaluation modes on the symex benchmark `numbrix-sat`, which has a single possible program path: additional threads lead to more overhead, with no performance gain (there are no branches for symbolic execution to explore in parallel). Second, on the symex benchmark `sort-7`, compile achieves higher speedups than compile-eager. On this benchmark, compile-eager’s SMT solving CPU times are about twice as fast as compile’s; nonetheless, its overall times are slightly slower. This benchmark generates many (28 million) derived tuples, and it could be that eager evaluation’s approach of generating one work item per tuple introduces too much overhead. This result suggests the potential of a hybrid evaluation scheme that uses some amount of batching to reduce overhead, while still achieving better SMT solver locality than semi-naive evaluation.

## 6 Formulog Performance: Present and Future

This section first contextualizes the performance of new-and-improved Formulog relative to non-Datalog reference implementations (Section 6.1), and then proposes future directions (Section 6.2).

### 6.1 Putting Formulog’s Performance in Context

In the original Formulog paper, Bembek et al. [2020b] compare the performance of Formulog to reference implementations that perform analyses similar to the case studies, but are written in traditional languages ( $F^\sharp$ , Java, and C++). While these comparisons are not apples-to-apples (implementations might use different heuristics, SMT encodings, etc.), they do provide wider context for Formulog’s performance. How does Formulog stack up now given compilation and eager evaluation? The numbers are promising: our performance improvements to Formulog substantially

Table 2. On 22/23 benchmarks, a compiled Formulog mode beats any Formulog interpreter mode; on 20/23 benchmarks, some Formulog mode beats the non-Datalog reference implementation (see Section 6.1; for the symex case study, we compare against KLEE [Cadar et al. 2008]). The best Formulog time on a benchmark is *italicized*; the best overall time for a benchmark is in **bold**. Timeouts were 30 minutes. In the case that a Formulog configuration timed out on its first trial, we did not run additional trials. SMT-heavy case studies are shaded gray .

Case study	Benchmark	Formulog interp. (s)		Formulog compile (s)		Reference impl. (s)
		semi-naive	eager	semi-naive	eager	
dminor	all-1	4.76	2.46	4.10	<b>0.60</b>	1.50
dminor	all-10	17.05	4.19	5.38	<b>1.40</b>	68.00
dminor	all-100	128.00	94.56	102.02	<b>74.05</b>	TO
scuba	antlr	310.38	TO	<b>93.89</b>	1015.10	95.11
scuba	avro	387.19	TO	<i>137.79</i>	TO	<b>85.16</b>
scuba	hedc	249.55	TO	<b>65.00</b>	272.36	81.01
scuba	hsqldb	244.84	TO	<b>64.38</b>	431.53	68.37
scuba	luindex	1338.49	TO	<b>1045.64</b>	TO	error
scuba	polyglot	243.34	1246.84	<b>56.66</b>	164.78	75.15
scuba	sunflow	1041.81	TO	<i>495.98</i>	TO	<b>279.81</b>
scuba	toba-s	257.02	TO	<b>75.21</b>	312.01	76.63
scuba	weblech	286.23	TO	<b>77.66</b>	799.52	97.22
scuba	xalan	285.88	TO	<i>84.56</i>	1057.24	<b>72.64</b>
symex	interp-5	117.64	6.80	121.24	<b>4.18</b>	46.81
symex	interp-6	541.73	16.93	523.55	<b>14.23</b>	176.50
symex	numbrix-sat	19.84	28.72	<b>11.87</b>	26.79	302.25
symex	numbrix-unsat	16.79	23.02	<b>10.13</b>	22.84	160.03
symex	prioqueue-5	39.72	7.89	39.81	<b>5.33</b>	41.11
symex	prioqueue-6	128.35	24.48	114.72	<b>19.91</b>	199.34
symex	shuffle-4	3.06	2.25	1.62	<b>0.64</b>	49.80
symex	shuffle-5	6.30	3.79	4.46	<b>2.06</b>	TO
symex	sort-6	15.04	9.62	10.80	<b>9.11</b>	140.20
symex	sort-7	112.56	<b>76.25</b>	76.77	82.02	1607.07

close the gap in cases where Formulog previously significantly trailed the reference implementations, and translate to increased speedups over the reference implementations in other cases (Table 2).

Except for the Dminor reference implementation, we ran the reference implementations on an m5.12xlarge AWS EC2 instance, using the same version of Z3 as in our Formulog experiments. The Dminor reference implementation can run only on an old version of the .NET platform, constrained both in terms of available memory and available Z3 versions; instead of running the Dminor reference implementation again, we use the numbers provided by [Bembenek et al. \[2020b\]](#).

*Refinement Type Checking (dminor)*. The dminor case study has three benchmarks, consisting, respectively, of one, 10, and 100 copies of the same skeleton Dminor program  $D_{all}$  (a composite of all publicly available Dminor programs that use only the core feature set). The reference type checker of [Bierman et al. \[2012\]](#) is written in F<sup>#</sup> and uses an optimization not implemented in the Formulog version; nonetheless, Formulog compile-eager consistently beats it. The reference implementation takes 1.5 seconds to type check one copy of  $D_{all}$ ; in interpret mode, Formulog takes 4.8 seconds; in

compile-eager mode, it takes 0.61 seconds. For the 10-copy version, the reference implementation takes 68 seconds, Formulog interpret takes 17 seconds, and Formulog compile-eager takes 1.4 seconds. The Formulog versions scale to type checking 100 copies of *Dall* (interpret: 128 seconds; compile-eager: 74 seconds); the reference implementation does not, timing out after 100 minutes (perhaps hamstrung by the old .NET platform it runs on).

*Bottom-Up Points-To Analysis (scuba).* The reference implementation of Feng et al. [2015], written in Java, achieves a mean  $3.5\times$  speedup over Formulog interpret (min/median/max:  $2.9\times/3.4\times/4.5\times$ ) on nine substantial Java benchmarks.<sup>13</sup> By compiling the Formulog version to Soufflé, we reduce this speedup to mean  $1.1\times$  (min / median / max:  $0.75\times / 0.98\times / 1.8\times$ ); that is, Formulog compile performs essentially on par with the reference implementation. The Formulog version closely follows the mathematical specification of the analysis, while integrating some of the heuristics used in the reference implementation, which is significantly more complex (and  $10\times$  larger in SLOC).

*Symbolic Execution (symex).* We compare against the symbolic executor KLEE (v3.0, released June 2023) [Cadaru et al. 2008], set to analyze LLVM 13.0 bitcode (the Formulog-based bounded symbolic executor analyzes LLVM 7.0 bitcode). Formulog interpret achieves a mean  $35\times$  speedup over KLEE (min / median / max:  $0.33\times / 9.4\times / 290\times$ ), an improved performance compared to the numbers reported in the original Formulog paper. Formulog compile-eager boosts this speedup up to mean  $100\times$  (min / median / max:  $7.0\times / 12\times / 870\times$ ). These numbers should be taken with a grain of salt: the Formulog-based symbolic executor only handles a small subset of LLVM, whereas KLEE is an industrial-strength tool. Although it implements a different analysis—C bounded model checking instead of LLVM symbolic execution—for the sake of completeness we also compare against the tool CBMC (v5.85.0, released June 2023) [Clarke et al. 2004]: it achieves a mean  $200\times$  speedup over Formulog compile-eager (min / median / max:  $0.26\times / 9.2\times / 890\times$ ). We explicitly set the number of times to unwind loops on two benchmarks to force CBMC to complete.

*Discussion.* The performance improvements developed in this paper were enabled by Formulog’s high-level design, a central desideratum being that additional features—functional programming, SMT solving, and relation reification—should not get in the way of evaluating and optimizing Formulog programs just as if they were Datalog programs. First, this allows a direct translation from Formulog to a high-performance Datalog platform, Soufflé. In the future, we could translate to different Datalog engines (e.g., Differential Datalog [Ryzhyk and Budiu 2019]), as long as they have an FFI and provide some control over the order in which predicates in a rule are evaluated (well typedness in Formulog depends on predicate order; our implementations of eager evaluation account for this). Second, it means that we can develop new *Formulog* evaluation algorithms by developing new *Datalog* evaluation strategies; e.g., by developing eager Datalog evaluation, we get eager Formulog evaluation. Compared to designing Formulog evaluation algorithms directly, this has the advantage that Datalog has fewer moving parts (and is thus easier to reason about) and is extensively studied. It also enabled us to relatively easily add eager evaluation to Soufflé, as we were able to reuse much of the Soufflé codebase, and did not have to make Soufflé aware of Formulog semantics.

<sup>13</sup>We omit the benchmark luindex, on which the reference implementation of Feng et al. [2015] fails with an out-of-bounds access. Bembenek et al. [2020b] report that the reference implementation completes on this benchmark in 100 seconds. The experimental setups differ in the implementation of the JDK 7 that is analyzed as part of the points-to analyses; we believe this leads to the discrepancy here.



## 6.2 Future Directions

*Hybrid Evaluation Schemes.* There is a spectrum between semi-naive and eager evaluation strategies. On the one hand, it is possible to make eager evaluation a little less eager by introducing some degree of batching; for example, the facts derived during the evaluation of a rule could always be grouped together into fixed-size batches. On the other hand, it is also possible to use eager evaluation and semi-naive evaluation within the same program; for instance, eager evaluation could be used for Formulog strata that contain SMT calls, and semi-naive evaluation could be used elsewhere. Furthermore, although it would require a bit more coordination, eager and semi-naive evaluation can be combined in the same stratum, so that different approaches are used for different rules. Hybrid evaluation schemes would be particularly relevant when integrating a component written in Formulog (and compiled to Soufflé) into an existing Soufflé project: one might want to use eager evaluation just for the Formulog part, and semi-naive evaluation everywhere else.

*More Sophisticated Distribution of SMT Queries.* Formulog currently associates a single SMT solver with each Datalog evaluation thread, and discharges all SMT queries arising on that thread to that solver. This tightly binds the distribution of SMT queries across SMT solvers to the distribution of inference tasks across Datalog evaluation threads. While this makes it possible to achieve more favorable distributions of the SMT workload just by changing the Datalog evaluation algorithm (viz. eager evaluation), it is also a limitation, as it rules out other possible ways of distributing the SMT workload. For example, one could imagine having a *shared* pool of SMT solvers: each Datalog worker thread submits queries to the pool, and the pool decides the “best” solver to run it on (perhaps based on the state of the solver, or some characteristic of the query). The pool might choose to allocate multiple cores/solvers to particularly “hard” queries using a portfolio approach [Wintersteiger et al. 2009]. However, even with more sophisticated schemes for distributing SMT queries, eager evaluation would still be useful, since it is ultimately the Datalog evaluation strategy that determines the order in which SMT queries are produced.

*Applying Eager Evaluation Beyond Formulog.* Eager evaluation’s DFS of the logical inference space could be beneficial to Datalog systems (beyond Formulog) that interact with stateful external systems whose performance depends on the distribution and order of queries. That interaction might be through a general foreign function interface or via specialized, built-in support. For example, Soufflé supports relations stored in external SQLite databases. The current implementation of Soufflé eagerly reads the entire relation into memory during initialization, but an alternate implementation might read only a slice of the database relation into memory at any one time (e.g., if the relation is too large to fit into memory); in this case, performance depends on the temporal and spatial locality of database queries. Another example is Vieira [Li et al. 2024], a probabilistic variant of Datalog with built-in predicates that discharge queries to external foundation models. Vieira currently interacts with foundation models using only one-off (context-free) calls; in an alternate model of neurosymbolic programming, Vieira could use stateful APIs for foundation models (such as OpenAI’s Assistants API<sup>14</sup>) so that each foundation model query is part of the same conversation. In this case, the way a foundation model answers a question would depend on the flow of the conversation up to that point—a function of how the Datalog inference space has been explored.

## 7 Related Work

*Pipelined Semi-Naive Evaluation.* Eager evaluation is similar in spirit to pipelined semi-naive evaluation (PSN), introduced by Loo et al. [2006] for Datalog-based declarative networking. Eager evaluation and PSN are both worklist-based algorithms that evaluate rules using one “delta” tuple

<sup>14</sup><https://platform.openai.com/docs/assistants/overview>

at a time. However, PSN is designed for a distributed, multi-node setting, whereas eager evaluation is designed for a single-node, multi-threaded setting. Despite its distributed setting, PSN is crafted so that rule evaluation is localized, making it easy to prove the algorithm’s completeness; in contrast, eager evaluation does not restrict which thread a computation is performed on, leading to data races and a subtle completeness argument. PSN timestamps facts to avoid doing redundant work compared to traditional semi-naive evaluation; eager evaluation could do something similar, although we suspect this would lead to unacceptable overhead on the workloads Formulog targets.

*Executing Datalog Variants.* Non-distributed Datalog variants typically strive to work with semi-naive evaluation [Arntzenius and Krishnaswami 2020; Bembek et al. 2020b; Madsen et al. 2016; Sahebolamri et al. 2023, 2022; Szabó et al. 2021]. Because semi-naive evaluation is the most widely used evaluation strategy for Datalog, being able to support semi-naive evaluation is seen as validating the practicality of the variant (e.g., there is an entire POPL paper on semi-naive evaluation for Datafun [Arntzenius and Krishnaswami 2020]). Our work demonstrates that, even in a single-node setting, semi-naive evaluation is not the optimal evaluation strategy for every Datalog variant.

*Traversal Orders in Logic Programming.* We are not aware of prior work that performs a DFS of the logical inference space during bottom-up evaluation (except for PSN [Loo et al. 2006], which has this capability). However, top-down logic programming systems like Prolog typically explore SLD resolution trees using DFS [Gallier 1986]. Top-down evaluation proceeds from the head of a rule to the body—i.e., starts with a conclusion, and then searches for a justification. Bottom-up evaluation works the opposite way by combining known facts to make new inferences. Given a query to prove, the magic set transformation [Bancilhon et al. 1985; Beeri and Ramakrishnan 1991] rewrites a logic program so that bottom-up evaluation derives only facts relevant to proving that query (simulating top-down evaluation); while this limits which logical inferences are made, those inferences are still made in a breadth-first order (assuming traditional semi-naive evaluation). In principle, bottom-up evaluation algorithms could pursue traversals through the logical inference space different from breadth-first and depth-first search. For example, an evaluation strategy that assigns a priority to each derived fact, and processes facts in this order, could be used to guide potentially non-terminating computations—like unbounded symbolic execution [Cadar et al. 2008] or enumerative program synthesis [Alur et al. 2013]—towards fruitful directions.

*Datalog Compilation.* Compilation is a popular strategy for speeding up Datalog and its variants. Soufflé [Jordan et al. 2016; Scholz et al. 2016] spearheaded this approach by compiling Datalog to an abstract instruction set that is then compiled to C++. The Flix implementation [Madsen et al. 2022] compiles Horn clauses to an abstract machine; Differential Datalog [Ryzhyk and Budiu 2019], Crepe [Zhang 2020], and Ascent [Sahebolamri et al. 2022] compile them to Rust (the latter two using macro-based approaches). Pacak and Erdweg [2022] compile Functional InCA—a functional language interpreted under a non-standard, fixpoint semantics—to Datalog. Our approach (Section 3) is hybrid, as we compile Formulog Horn clauses to traditional Datalog, but Formulog functional code directly to C++. Flan [Abeyasinghe et al. 2024] is a new compilation framework for Datalog that leverages Lightweight Modular Staging [Rompf and Odersky 2010] to flexibly produce high performing code. As future work, the Flan authors suggest using Flan to construct something akin to a “compiled Formulog.” We are excited about this direction, as Flan’s flexibility would make it easier to combine Formulog with additional language features (like lattices). Our work suggests that Flan would need to support eager evaluation to achieve top performance on SMT-heavy workloads.

## 8 Summary

This paper has explored both off-the-shelf technologies and custom techniques to speed up Formulog, a domain-specific language that combines Datalog, SMT solving, and functional programming. Compiling to off-the-shelf Soufflé provides solid (mean 2.2×) speedups; an additional advantage comes via *eager evaluation*, a novel strategy that structures Datalog evaluation as a worklist algorithm and achieves a quasi-depth-first search through the logical inference space by submitting inference tasks to a work-stealing thread pool. The key insight is that, for Formulog, the order in which you explore logical inferences matters, because different orders lead to different distributions of the SMT workload across threads. On SMT-heavy workloads, eager evaluation extensions to the Formulog interpreter and Soufflé’s code generator achieve mean 5.2× and 7.6× speedups, respectively, over off-the-shelf Soufflé. Our performance improvements to Formulog help make Formulog-based analyses competitive with previously published, non-Datalog analysis implementations, achieving faster times on 20 out of 23 benchmarks. Our results provide strong evidence that Formulog can be a realistic platform for SMT-based static analysis, and also add some nuance to the conventional wisdom that fast Datalog evaluation depends on semi-naïve evaluation.

## Data-Availability Statement

An artifact supporting the results of this paper is available on Zenodo [Bembenek et al. 2024b]. It includes our extensions to Formulog and Soufflé, the benchmarks we use, the data from our experiments, benchmarking and data-processing scripts, and documentation on how to build on top of our software. Additionally, Formulog is available at <https://github.com/HarvardPL/formulog>.

## Acknowledgments

We thank Eric Zhang for his contributions to a preliminary implementation of the Formulog-to-Soufflé compiler; we still use some of the code he wrote. We thank Eddie Kohler and the anonymous reviewers for helpful feedback on earlier versions of this paper. We also belatedly thank Yu Feng and Xinyu Wang for answering our questions about Scuba and giving us access to the reference Scuba implementation when we were writing the original Formulog paper (their assistance was mistakenly omitted from that paper’s acknowledgments).

## References

- Supun Abeysinghe, Anxhelo Xhebraj, and Tiark Rompf. 2024. Flan: An Expressive and Efficient Datalog Compiler for Program Analysis. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2577–2609. <https://doi.org/10.1145/3632928>
- Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An Overview of the Saturn Project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 43–48. <https://doi.org/10.1145/1251535.1251543>
- Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. <https://ieeexplore.ieee.org/document/6679385/>
- Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M Hellerstein, and Russell Sears. 2010a. Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the 5th European Conference on Computer Systems*. 223–236. <https://doi.org/10.1145/1755913.1755937>
- Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. 2010b. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - 1st International Workshop*. 43–48. [https://doi.org/10.1007/978-3-642-24206-9\\_16](https://doi.org/10.1007/978-3-642-24206-9_16)
- Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Soufflé: A Tale of Inter-Engine Portability for Datalog-Based Analyses. In *International Workshop on State Of the Art in Program Analysis*. 25–30. <https://doi.org/10.1145/3088515.3088522>
- Krzysztof R Apt, Howard A Blair, and Adrian Walker. 1988. Towards a Theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming*. Elsevier, 89–148. <https://doi.org/10.1016/B978-0-934613-40-8.50006-3>
- Michael Arntzenius and Neel Krishnaswami. 2020. Semi-naïve Evaluation for a Higher-Order Functional Language. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 22:1–22:28. <https://doi.org/10.1145/3371090>

- Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. 119–129.
- Francois Bancilhon. 1986. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems*. Springer, 165–178. [https://doi.org/10.1007/978-1-4612-4980-1\\_17](https://doi.org/10.1007/978-1-4612-4980-1_17)
- Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 1–15. <https://doi.org/10.1145/6012.15399>
- Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). [www.SMT-LIB.org](http://www.SMT-LIB.org).
- Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *The Journal of Logic Programming* 10, 3-4 (1991), 255–299. [https://doi.org/10.1016/0743-1066\(91\)90038-Q](https://doi.org/10.1016/0743-1066(91)90038-Q)
- Aaron Bembek, Michael Ballantyne, Michael Greenberg, and Nada Amin. 2020a. Datalog-Based Systems Can Use Incremental SMT Solving. In *Proceedings of the 36th International Conference Logic Programming (Technical Communications)*.
- Aaron Bembek, Michael Greenberg, and Stephen Chong. 2020b. Formulog: Datalog for SMT-Based Static Analysis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 141:1–141:31. <https://doi.org/10.1145/3428209>
- Aaron Bembek, Michael Greenberg, and Stephen Chong. 2024a. Making Formulog Fast: An Argument for Unconventional Datalog Evaluation (Extended Version). arXiv:2408.14017 [cs.PL]
- Aaron Bembek, Michael Greenberg, and Stephen Chong. 2024b. *Making Formulog Fast: An Argument for Unconventional Datalog Evaluation (OOPSLA'24 Artifact)*. <https://doi.org/10.5281/zenodo.13372573>
- Gavin M. Bierman, Andrew D. Gordon, Cătălin Hrițcu, and David Langworthy. 2012. Semantic Subtyping with an SMT Solver. *Journal of Functional Programming* 22, 1 (2012), 31–105. <https://doi.org/10.1145/1863543.1863560>
- Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 169–190. <https://doi.org/10.1145/1167473.1167488>
- Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.
- Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 243–262. <https://doi.org/10.1145/1639949.1640108>
- Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 209–224.
- Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What You Always Wanted to Know About Datalog (and Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering* 1, 1 (1989), 146–166.
- Keith L Clark. 1977. Negation as Failure. In *Logic and Data Bases*. 293–322. [https://doi.org/10.1007/978-1-4684-3384-5\\_11](https://doi.org/10.1007/978-1-4684-3384-5_11)
- Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 168–176. [https://doi.org/10.1007/978-3-540-24730-2\\_15](https://doi.org/10.1007/978-3-540-24730-2_15)
- Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering* 5, 1 (1998), 46–55.
- Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340. [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
- Leonardo de Moura and Nikolaj Bjørner. 2011. Satisfiability modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (2011), 69–77. <https://doi.org/10.1145/1995376.1995394>
- Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2006. Specifying and reasoning about dynamic access-control policies. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning*. 632–646. [https://doi.org/10.1007/11814771\\_51](https://doi.org/10.1007/11814771_51)
- Niklas Eén and Niklas Sörensson. 2003. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science* 89, 4 (2003), 543–560. [https://doi.org/10.1016/s1571-0661\(05\)82542-3](https://doi.org/10.1016/s1571-0661(05)82542-3)
- Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutiris, and Jignesh M. Patel. 2019. Scaling-up in-Memory Datalog Processing: Observations and Techniques. *Proceedings of the VLDB Endowment* 12, 6 (feb 2019), 695–708. <https://doi.org/10.14778/3311880.3311886>

- Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-up Context-Sensitive Pointer Analysis for Java. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems*. 465–484. [https://doi.org/10.1007/978-3-319-26529-2\\_25](https://doi.org/10.1007/978-3-319-26529-2_25)
- Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *Proceedings of the 29th USENIX Security Symposium*. 1075–1092.
- Hervé Gallaire and Jack Minker (Eds.). 1978. *Logic and Data Bases*. Plenum Press.
- Jean Gallier. 1986. *Logic for Computer Science: Foundations of Automated Theorem Proving*. Wiley, Chapter 9.
- Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*.
- Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering*. 1176–1186. <https://doi.org/10.1109/ICSE.2019.00120>
- Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 116:1–116:27. <https://doi.org/10.1145/3276486>
- Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Foundations & Trends in Databases* (2013). <https://doi.org/10.1561/19000000017>
- Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492. <https://doi.org/10.1145/78969.78972>
- Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. 2021. Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning. In *Advances in Neural Information Processing Systems*. 25134–25145.
- Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*. [https://doi.org/10.1007/978-3-319-41540-6\\_23](https://doi.org/10.1007/978-3-319-41540-6_23)
- Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019a. Brie: A specialized trie for concurrent Datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. 31–40.
- Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019b. A Specialized B-tree for Concurrent Datalog Evaluation. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 327–339. <https://doi.org/10.1145/3293883.3295719>
- Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2022. Specializing parallel data structures for Datalog. *Concurrency and Computation: Practice and Experience* 34, 2 (2022). <https://doi.org/10.1002/CPE.5643>
- Bas Ketsman and Paraschos Koutris. 2022. Modern Datalog Engines. *Foundations and Trends in Databases* 12, 1 (2022), 1–68. <https://doi.org/10.1561/19000000073>
- Ninghui Li and John C Mitchell. 2003. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*. 58–73. [https://doi.org/10.1007/3-540-36388-2\\_6](https://doi.org/10.1007/3-540-36388-2_6)
- Ziyang Li, Jiani Huang, Jason Liu, Felix Zhu, Eric Zhao, William Dodds, Neelay Velingker, Rajeev Alur, and Mayur Naik. 2024. Relational Programming with Foundation Models. In *AAAI Conference on Artificial Intelligence*. 209–214.
- Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A Language for Neurosymbolic Programming. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1463–1487. <https://doi.org/10.1145/3591280>
- Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 97–108. <https://doi.org/10.1145/1142473.1142485>
- Magnus Madsen, Jonathan Lindegaard Starup, and Ondřej Lhoták. 2022. Flix: A Meta Programming Language for Datalog. In *Proceedings of the 4th International Workshop on the Resurgence of Datalog in Academia and Industry*.
- Magnus Madsen, Ming-Ho Yee, and Ondřej 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*. 194–208. <https://doi.org/10.1145/2908080.2908096>
- Eric Mohr, David A Kranz, and Robert H Halstead Jr. 1990. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 185–197.
- Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The Magic of Duplicates and Aggregates. In *Proceedings of the 16th International Conference on Very Large Data Bases*. 264–277.
- André Pacak and Sebastian Erdweg. 2022. Functional programming with Datalog. In *36th European Conference on Object-Oriented Programming*. 7:1–7:28.
- Teodor C Przymusiński. 1988. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*. Elsevier, 193–216. <https://doi.org/10.1016/b978-0-934613-40-8.50009-9>

- Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *International Conference on Generative Programming and Component Engineering*. 127–136. <https://doi.org/10.1145/1868294.1868314>
- Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Proceedings of the 3rd International Workshop on the Resurgence of Datalog in Academia and Industry*. 56–67.
- Arash Sahebollahri, Langston Barrett, Scott Moore, and Kristopher K. Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1198–1223. <https://doi.org/10.1145/3622840>
- Arash Sahebollahri, Thomas Gilray, and Kristopher Micinski. 2022. Seamless deductive inference via macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 77–88.
- Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. 196–206. <https://doi.org/10.1145/2892208.2892226>
- Jiwon Seo, Stephen Guo, and Monica S. Lam. 2013. Socialite: Datalog Extensions for Efficient Social Network Analysis. In *29th IEEE International Conference on Data Engineering*. 278–289. <https://doi.org/10.1109/ICDE.2013.6544832>
- Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data*. 1135–1149. <https://doi.org/10.1145/2882903.2915229>
- Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.
- Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *Proceedings of the VLDB Endowment* 12, 2 (2018), 141–153.
- Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-Based Program Analyses in Datalog. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 139:1–139:29. <https://doi.org/10.1145/3276509>
- Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1–15.
- Petar Tsankov, Andrei Dan, Dana Drachler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82. <https://doi.org/10.1145/3243734.3243780>
- Allen Van Gelder. 1989. Negation as Failure Using Tight Derivations for General Logic Programs. *The Journal of Logic Programming* 6, 1-2 (1989), 109–133. [https://doi.org/10.1016/0743-1066\(89\)90032-0](https://doi.org/10.1016/0743-1066(89)90032-0)
- 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*. 97–118.
- John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. 131–144. <https://doi.org/10.1145/996841.996859>
- Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. 2009. A Concurrent Portfolio Approach to SMT Solving. In *Computer Aided Verification*. 715–720.
- Eric Zhang. 2020. Crepe. <https://github.com/ekzhang/crepe>. Accessed: 2023-08-18.

Received 2024-04-03; accepted 2024-08-18