Formulog: Datalog + SMT + FP

Aaron Bembenek[1], Michael Greenberg[2], and Stephen Chong[1]

[1] Harvard University, Cambridge, MA 02138, USA,
bembenek@g.harvard.edu, chong@seas.harvard.edu
[2] Stevens Institute of Technology, Hoboken, NJ 07030, USA,
michael@greenberg.science

Abstract. Formulog extends Datalog with mechanisms for constructing logical terms and reasoning about them with satisfiability modulo theories (SMT) solving; a first-order functional language aids inspecting and manipulating complex terms. This combination of features makes it possible to write complex SMT-based program analyses in a way close to their formal specification, while being satisfactorily performant thanks to powerful Datalog optimizations and efficient evaluation techniques.

Keywords: Datalog, satisfiability modulo theories, program analysis

1 Introduction

Formulog [3] is a domain-specific language for writing program analyses that use satisfiability modulo theories (SMT) solving. Formulog extends Datalog with a first-order functional language, ergonomic syntax for writing SMT terms, and a robust interface to an SMT solver. Datalog has already been shown to be a useful language for writing certain static analyses [12, 9, 8]; however, many static analyses cannot be easily written in existing Datalog variants, as they require SMT solvers to reason about logical formulas containing constructs like arrays and bit vectors. Formulog fills this gap and meets several key design objectives.

First, *Formulog enables executable specifications.* Its combination of features enables programming SMT-based static analyses close to their mathematical specifications, which typically consist of inference rules and pure functions, either of which might need to check the satisfiability or validity of logical terms.

Second, *Formulog lets Datalog be Datalog.* Despite the additional language features, Formulog programs benefit from Datalog optimizations and evaluation techniques (i.e., semi-naive evaluation). This makes it possible to write analyses at the level of a mathematical specification but still get solid performance.

Third, *Formulog's parts interact in a safe and expressive way.* The interaction between Datalog, functions, and SMT solving needs to be safe, never raising type errors by, e.g., querying the satisfiability of a term that is not well sorted under SMT. At the same time, it needs to be possible to construct expressive SMT formulas. We strike this balance through a bimodal type system that treats terms occurring in SMT formulas more liberally than terms outside of formulas.

We first present Formulog by example (Section 2); we then discuss our prototype and several case studies (Section 3).

2 Formulog in a Nutshell

We present Formulog by example. The canonical “hello world” program for Datalog computes graph transitive closure. An analogous problem for Formulog is computing non-empty paths in a proposition graph, i.e., a directed graph where edges are labeled with SMT propositions, where we consider a path to exist only if the conjunction of the propositions along it (the path condition) is satisfiable. Consider the example graph in Figure 1, which has edges labeled with signed comparisons between symbolic 32-bit vectors (like the proposition $x < y$, where x and y are SMT variables of the 32-bit vector sort). There is no path from node 2 to node 1, as no sequence of edges leads from node 2 to node 1. There is also no *satisfiable* path from node 1 to node 4. While a path exists structurally, the conjunction of the edge conditions—the path condition $x < y \land y < z \land z < x$—is not satisfiable. However, there is a (non-empty) path from node 2 to itself, since the path condition $y < z \land z < x \land y > z + x$ is satisfiable under the theory of bit vectors (the addition of x and y can wrap, unlike the theory of mathematical integers). We break down the Formulog program computing non-empty paths on the example proposition graph piece-by-piece (Figure 2):

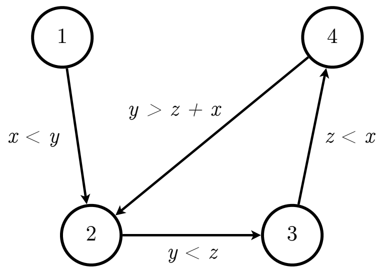

Fig. 1. A proposition graph is a directed graph where edges are labeled with SMT propositions. The operators < and > are signed comparisons between 32-bit vectors.

Lines 1-2. We declare `edge` as a ternary input (EDB) relation. Formulog is strongly typed: `edge` relates two terms of type `node` (an alias for `i32`, a 32-bit vector) and a term of type `bool smt`, i.e., an SMT proposition.

Lines 4-7. We enumerate the `edge` relation for our example graph. The third attribute is the SMT proposition. SMT terms are complex terms built from a library of constructors like `bv_slt`, which represents signed bit vector less-than; it constructs a term of type `bool smt` out of two terms of type `bv[`k`] smt` (where k is a positive integer; `i32` is shorthand for `bv[32]`). Quasiquoting (with backticks) enables a type checking mode that flexibly mixes concrete and SMT terms. The notation `#`x`[`τ`]` denotes an SMT variable of sort τ named x.

Lines 9-10. Formulog supports first-order, polymorphic, recursive ML-style functions. Here the `mem` function checks list membership. Such functions are syntactic sugar, and could be translated into (less efficient) Datalog; functions are convenient for inspecting and manipulating complex terms. We use ML shorthand for list constructors (t `::` t and `[`t`, ..., `t`]` for terms t).

Line 12. We declare the `path` output (IDB) relation, relating two nodes, a path (a list of nodes), and a path condition (an SMT proposition).

Lines 13-15. The first rule defining `path` is the non-recursive case: there is a path between two nodes if there is an edge between them labeled with a

```
1  type node = i32
2  input edge(node, node, bool smt)
3
4  edge(1, 2, `bv_slt(#x[i32], #y[i32])`).
5  edge(2, 3, `bv_slt(#y[i32], #z[i32])`).
6  edge(3, 4, `bv_slt(#z[i32], #x[i32])`).
7  edge(4, 2, `bv_sgt(#y[i32], bv_add(#z[i32], #x[i32]))`).
8
9  fun mem(x: 'a, xs: 'a list) : bool =
10   match xs with [] => false | h :: t => x = h || mem(x, t) end
11
12 output path(node, node, node list, bool smt)
13 path(X, Y, [Y], Phi) :-
14   edge(X, Y, Phi),
15   is_sat(Phi) = true.
16 path(X, Z, Y :: Path, `Phi /\ Constraint`) :-
17   edge(X, Y, Phi),
18   path(Y, Z, Path, Constraint),
19   mem(Y, Path) = false,
20   is_sat(`Phi /\ Constraint`) = true.
21
22 output path_conditions(node, node, bool smt list)
23 path_conditions(X, Y, Conditions) :-
24   path(X, Y, _, _),
25   Conditions = path(X, Y, _, ??).
```

Fig. 2. This Formulog program computes non-empty paths (the `path` relation) through our example proposition graph (Figure 1), ignoring paths where the accumulated condition along the path is not satisfiable. It also aggregates all the path conditions for each pair of nodes that have a path between them (the `path_conditions` relation).

satisfiable formula. Datalog variables have initial capitals. The function `is_sat` takes a term of type `bool smt` and returns a `bool` indicating the satisfiability of its argument.

Lines 16-21. The second rule defining `path` is the recursive case; it makes use of both the user-defined function `mem` and the built-in function `is_sat`.

Lines 22-26. We declare a relation `path_conditions` relating two nodes and a list of the path conditions for paths between them. This relation's sole rule takes advantage of Formulog's flexible approach to stratified aggregation: the notation `path(X, Y, _, ??)` treats the relation `path` as a function that takes two arguments (grouping variables `X` and `Y`) and returns a list of all the elements in the fourth column—i.e., the path conditions (the term `??` is a wildcard). In addition to stratified aggregation, Formulog supports stratified negation.

This combination of features—and Formulog's take on them—distinguishes Formulog from other systems combining logic programming and constraint solving, like Calypso [1] and constraint logic programming [10].

3 Prototype and Case Studies

We implemented a prototype of Formulog in ∼24K lines of Java.[3] The prototype is designed as a relatively standard Datalog engine that discharges SMT queries to external SMT solvers; it uses caching to take advantage of incremental SMT solving [2]. It supports optimizations like the magic set transformation (which can be applied on a relation-by-relation basis) and automatic parallelization. We developed three substantial case studies with Formulog to demonstrate (a) that complex static analysis specifications encode naturally into Formulog, and (b) that Datalog optimizations can help the resulting programs achieve satisfactory performance (at times besting reference implementations).[4]

Refinement Type Checker (1.2K LOC). This type checker uses SMT solving to prove subtyping between refinement types, which requires checking logical validity. Our implementation translates the original formalism [4] almost line-by-line; our encoding exposed an important bug in the formalism. Thanks to automatic parallelization, it scales better than the reference implementation.

Java Pointer Analysis (1.5K LOC). SMT solving helps statically resolve the memory locations Java variables might point to; this is a direct translation of the formal specification of Feng et al. [7] (it uncovered bugs in the specification). It is slower than the reference implementation, but much smaller (10% of the size), and thus might be appropriate as a prototype, as well as a playground for testing new features (like parallelizing the analysis or making it goal-directed).

LLVM Symbolic Evaluator (1K LOC). This tool evaluates a fragment of LLVM bitcode on symbolic inputs, up to a bounded depth. When it reaches a branch conditioned on a symbolic value, it explores each branch (assuming both are possible given the path so far). By writing the symbolic executor in Formulog, we can *automatically* parallelize exploration; by using the magic set transformation, we can guide the symbolic executor to avoid uninteresting paths. These optimizations help it compete with industrial-strength CBMC [6] and KLEE [5] on sample problems (with speedups of up to 12× over KLEE).

4 Outlook

We have shown that Formulog's combination of Datalog, SMT solving, and first-order functional programming makes it an appropriate language for coding up complex SMT-based program analyses. Even though our prototype Formulog runtime is not heavily optimized, it achieves satisfactory performance on our case studies, even in comparison to heavily optimized systems. In the future, we hope to improve its performance via compilation, à la Soufflé [11].

We also intend to expand Formulog's applications beyond being a language for writing program analyses. One direction is to use Formulog as an *intermediate verification language* (i.e., an analysis produces Formulog code, instead of being

[3] Available at https://github.com/HarvardPL/formulog.
[4] Available at https://zenodo.org/record/4039122.

written in Formulog). In particular, we are currently exploring the possibility of symbolically executing Datalog programs by translating them to Formulog programs. Furthermore, while we have focused on program analysis applications because we are most familiar with them, we are hopeful that Formulog will prove useful in other domains, such as knowledge representation and reasoning.

Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-19-C-0004. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

References

1. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the Saturn project. In: Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools and Eng., pp. 43–48 (2007)
2. Bembenek, A., Ballantyne, M., Greenberg, M., Amin, N.: Datalog-based systems can use incremental SMT solving. In: Proc. 36th Int. Conf. Log. Program. (2020)
3. Bembenek, A., Greenberg, M., Chong, S.: Formulog: Datalog for SMT-based static analysis. Proc. ACM Program. Languages **4**(OOPSLA), 141:1–141:31 (2020)
4. Bierman, G.M., Gordon, A.D., Hriţcu, C., Langworthy, D.: Semantic subtyping with an SMT solver. J. Functional Program. **22**(1), 31–105 (2012)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. 8th USENIX Conf. Operating Syst. Des. and Implementation, pp. 209–224 (2008)
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. 10th Int. Conf. Tools and Algorithms Construction and Anal. Syst., pp. 168–176 (2004)
7. Feng, Y., Wang, X., Dillig, I., Dillig, T.: Bottom-up context-sensitive pointer analysis for java. In: Proc. 13th Asian Symp. Program. Languages and Syst., pp. 465–484 (2015)
8. Flores-Montoya, A., Schulte, E.: Datalog disassembly. In: 29th USENIX Secur. Symp., pp. 1075–1092 (2020)
9. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. Proc. ACM Program. Languages **2**(OOPSLA), 116:1–116:27 (2018)
10. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proc. 14th ACM SIGACT-SIGPLAN Symp. Princ. Program. Languages, pp. 111–119 (1987)
11. Jordan, H., Scholz, B., Subotić, P.: Soufflé: On synthesis of program analyzers. In: Proc. 28th Int. Conf. Comput. Aided Verification, pp. 422–430 (2016)
12. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proc. ACM SIGPLAN 2004 Conf. Program. Lang. Des. and Implementation, pp. 131–144 (2004)