Formulog: Datalog for SMT-Based Static Analysis

AARON BEMBENEK, Harvard University, USA
MICHAEL GREENBERG*, Pomona College, USA
STEPHEN CHONG, Harvard University, USA

Satisfiability modulo theories (SMT) solving has become a critical part of many static analyses, including symbolic execution, refinement type checking, and model checking. We propose Formulog, a domain-specific language that makes it possible to write a range of SMT-based static analyses in a way that is both close to their formal specifications and amenable to high-level optimizations and efficient evaluation.

Formulog extends the logic programming language Datalog with a first-order functional language and mechanisms for representing and reasoning about SMT formulas; a novel type system supports the construction of expressive formulas, while ensuring that neither normal evaluation nor SMT solving goes wrong. Our case studies demonstrate that a range of SMT-based analyses can naturally and concisely be encoded in Formulog, and that — thanks to this encoding — high-level Datalog-style optimizations can be automatically and advantageously applied to these analyses.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Domain specific languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: Datalog, SMT solving

ACM Reference Format:
Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 141 (November 2020), 31 pages. https://doi.org/10.1145/3428209

1 INTRODUCTION

Satisfiability modulo theories (SMT) solving provides a way to reason logically about common program constructs such as arrays and bit vectors, and as such has become a key component of many static analyses. For example, symbolic execution tools use SMT solving to prune infeasible execution paths [Cadar et al. 2008; Cadar and Sen 2013]; type checkers use it to prove subtyping relations between refinement types [Bierman et al. 2012; Rondon et al. 2008]; and model checkers use it to abstract program states [Cimatti and Griggio 2012; McMillan 2006]. This paper presents Formulog, a domain-specific language for writing SMT-based static analyses. Formulog makes it possible to concisely encode a range of SMT-based static analyses in a way that is close to their formal specifications. Furthermore, Formulog is designed so that analyses implemented in it are amenable to efficient evaluation and powerful, high-level optimizations, including parallelization and automatic transformation of exhaustive analyses into goal-directed ones.

Formulog is based on Datalog, a logic programming language used to implement static analyses ranging from points-to analyses [Bravenboer and Smaragdakis 2009; Whaley and Lam 2004] to

*Work done while on sabbatical at Harvard University.

Authors' addresses: Aaron Bembenek, Harvard University, USA, bembenek@g.harvard.edu; Michael Greenberg, Pomona College, USA, michael@cs.pomona.edu; Stephen Chong, Harvard University, USA, chong@seas.harvard.edu.

2475-1421/2020/11-ART141
https://doi.org/10.1145/3428209

decompilers [Flores-Montoya and Schulte 2020; Grech et al. 2019] to security analyses [Grech et al. 2018; Guarnieri and Livshits 2009; Jordan et al. 2016; Livshits and Lam 2005; Tsankov et al. 2018]. Embodying the principle of separating the logic of a computation from the control necessary to perform that computation [Kowalski 1979], Datalog frees analysis designers from low-level implementation details and enables them to program at the level of specifications (such as formal inference rules). This leads to concise implementations [Whaley et al. 2005] that can be easier to reason about and improve at the algorithmic level compared to analyses in more traditional languages [Smaragdakis and Bravenboer 2011]. Datalog-based analyses can be fast and scalable, even outperforming the non-Datalog state-of-the-art [Bravenboer and Smaragdakis 2009]. Indeed, Datalog's high-level nature makes it amenable to high-level optimizations, such as parallelization [Scholz et al. 2016] and synthesis of goal-directed analyses from exhaustive ones [Reps 1995].

However, despite the appeal of Datalog for static analysis and the importance of SMT solving in static analysis, until now there has not been a focused study of how to effectively extend the benefits of Datalog to SMT-based analyses; our work bridges this gap.

Formulog augments Datalog with an interface to an external SMT solver and a first-order fragment of the functional language ML. It provides a library of constructors for building terms that are interpreted as logical formulas when applied to special SMT operators; in the backend, these operators are implemented by calls to an external SMT solver. A Formulog program is essentially a set of ML-style function definitions and Datalog-style rules; both pieces can refer to each other and invoke the SMT operators. As in Datalog, the goal of Formulog evaluation is to compute all possible inferences with respect to the rules, which correspond to logical implications. Unlike Datalog, rule evaluation might involve both ML evaluation and calls to an SMT solver.

The way this design combines Datalog, ML, and SMT solving gives Formulog some desirable properties. First, Formulog programs can use SMT solving the way it is used in SMT-based analyses. This results from the choice to represent SMT formulas as ML terms, and contrasts with the approach of most prior work combining logic programming and constraint solving (where, e.g., checking for formula validity is hard). Second, the combination of Datalog-style rules and ML-style functions mirrors the combination of inference rules and helper functions commonly used in analysis specifications, making it easier to translate formal analysis specifications into executable code. This close correspondence between specification and implementation means that specification-level reasoning is still applicable to analysis implementations (and vice versa: unexpected behavior in Formulog programs has revealed bugs in specifications). Third, because Formulog is based on Datalog, analyses written in it can be effectively optimized and evaluated via powerful Datalog algorithms, making them competitive with analyses written in more mature languages.

It takes care to fit Datalog, ML, and SMT solving together in a way that truly achieves these properties. Along these lines, part of our technical contribution is a novel bimodal type system that treats terms appearing in SMT formulas more liberally than terms appearing outside of formulas, making it possible to construct expressive logical formulas, while still ensuring that neither concrete (i.e., Datalog/ML) evaluation nor SMT solving goes wrong.

To test the practicality of Formulog, we implemented a fully featured, prototype Formulog runtime and wrote three substantial SMT-based analyses in Formulog: a type checker for a refinement type system, a bottom-up points-to analysis for JVM bytecode, and a bounded symbolic evaluator for a subset of LLVM bitcode. Our implementations for the first two case studies are almost direct translations of previously published formal specifications [Bierman et al. 2012; Feng et al. 2015]; indeed, Formulog allowed us to program close enough to the specifications to uncover bugs in both of them. Despite encoding complex analysis logic, each of our analyses is concise (no more than 1.5K LOC). Furthermore, our Formulog-based implementations have acceptable performance, even

Programs	prog	::=	H^*	Variables	X	$\in$	Var
Horn clauses	H	::=	$p(e^*)$:– P^*	Constructors	c	$\in$	CtorVar
Premises	P	::=	$A \mid !A$	Predicates	p	$\in$	PredVar
Atoms	A	::=	$p(e^*) \mid e = e$				
Expressions	e	::=	$X \mid c$				

Fig. 1. A Datalog program is a collection of Horn clauses that represent rules for making inferences.

when compared against reference implementations running on more mature language platforms. In some cases, we actually achieve substantial speedups over the reference implementations.

These performance results are possible only because Formulog's design allows our runtime to automatically and effectively apply high-level optimizations to Formulog programs. Our third case study makes this point emphatically. Due to automatic parallelization, our symbolic evaluator achieves a speedup of 8× over the symbolic execution tool KLEE [Cadar et al. 2008]. Moreover, this speedup increases to 12× when we use the magic set transformation [Bancilhon et al. 1985; Beeri and Ramakrishnan 1991] to automatically transform our exhaustive symbolic evaluator into a goal-directed one that explores only paths potentially leading to assertion failures. That Datalog can speed up analyses like points-to analysis is well established [Bravenboer and Smaragdakis 2009; Whaley and Lam 2004]; that it can automatically scale symbolic evaluation is a novel result.

In sum, this paper makes the following contributions:

- the design of Formulog (Section 3), a domain-specific language for writing SMT-based static analyses that judiciously combines Datalog, a fragment of ML, and SMT solving;
- a lightweight bimodal type system (Section 4) that mediates the interface between concrete evaluation and SMT solving, enabling the construction of expressive formulas while preventing many kinds of runtime errors in both concrete evaluation and SMT solving;
- a fully featured prototype and three substantial case studies (Section 5), showing that the design of Formulog can be the basis of a practical tool for writing SMT-based analyses; and
- an evaluation of Formulog's design in light of these case studies (Section 6), demonstrating how careful design decisions make Formulog an effective medium for encoding a range of SMT-based analyses in a way that is both close to their formal specifications and amenable to efficient evaluation and high-level optimizations.

2 BACKGROUND

The starting point for Formulog is Datalog with stratified negation (Figure 1) [Apt et al. 1988; Gallaire and Minker 1978; Green et al. 2013; Przymusinski 1988; Van Gelder 1989]. A Datalog program is a collection of Horn clauses, where a *clause* H consists of a head predicate $p(e^*)$ and a sequence of body premises P. Each *premise* P is either a positive atom A or a negated atom $!A$. An *atom* A has one of two forms: It is either a predicate symbol applied to a list of expressions, or the special equality predicate $e = e$. An *expression* e is a variable X or a nullary constructor c, i.e., an uninterpreted constant. Each predicate symbol p is associated with an extensional database (EDB) relation or an intensional database (IDB) relation. An *EDB relation* is tabulated explicitly through *facts* (clauses with empty bodies), whereas an *IDB relation* is computed through *rules* (clauses with non-empty bodies). A rule should be read as a universally quantified logical implication, with the conjunction of the body premises implying the predicate in the head. Datalog evaluation amounts to computing every possible inference with respect to these implications; the restriction to stratified

negation (a relation cannot be defined, either directly or indirectly, by its complement) ensures that this can be done via a sequence of fixed point computations.

Datalog has proven to be a natural and effective way to encode a range of static analyses [Bravenboer and Smaragdakis 2009; Flores-Montoya and Schulte 2020; Grech et al. 2019, 2018; Guarnieri and Livshits 2009; Jordan et al. 2016; Livshits and Lam 2005; Tsankov et al. 2018; Whaley and Lam 2004]. EDB relations are used to represent the program under analysis; for example, EDB relations might encode a control flow graph (CFG) of the input program. The logic of the analysis is encoded using rules that define IDB relations; these rules are fixed and do not depend on the program under analysis (which is already captured by the EDB relations). The Datalog program will compute the contents of the IDB relations, which can be thought of as the analysis results.

That being said, standard Datalog is a very restricted language and there are many other analyses that cannot easily be encoded in it, if at all. Recent variants extend Datalog for analyses that operate over interesting lattices [Madsen et al. 2016; Szabó et al. 2018]. Following in this spirit, Formulog proposes a way to support analyses that need access to SMT solving.

3 LANGUAGE DESIGN

The design of Formulog is driven by three main desiderata. First, it should be possible to implement SMT-based static analyses in a form close to their formal specifications. Second, it should be easy to use logical terms the way that they are commonly used in many analyses. For example, analyses often need to create formulas about entities such as arrays and machine integers, test those formulas for satisfiability or validity, and generate models of them. Third, Formulog programs should still be amenable to powerful Datalog optimizations and evaluable using scalable Datalog algorithms.

Section 6 demonstrates how the design of Formulog largely meets these desiderata. Here, we give a warm-up example of Formulog, provide an overview of its language features, discuss how these features support logical formulas, and conclude with its operational semantics.

3.1 Formulog by Example

To give the flavor of Formulog-based analyses, this section presents a bounded symbolic evaluator for CFGs of a simple imperative language (Figures 2 and 3). A symbolic evaluator [King 1976] interprets a program in which some values are unknown. When the evaluator reaches a condition that depends on one of these symbolic values, it forks into two processes, one in which the condition is assumed to be true and one in which it is assumed to be false. At this point, it can avoid exploring an impossible path by checking whether the condition along that branch is consistent with the conditions encountered so far during execution (which are known collectively as the "path condition"). Our symbolic evaluator uses fuel to bound the depth of its execution.

We use algebraic data types to represent the input language to the evaluator (lines 1-9). Values are formulas representing 32-bit vectors (i.e., terms of type `i32 smt`), and operands are either values or variables. Our input language has binary operations, conditional jumps, and fail instructions (indicating that control flow has reached, e.g., an assertion failure). The program-to-analyze is given by two EDB (i.e., `input`) relations (lines 11-12). The relation `node_has_inst` maps a CFG node to the corresponding instruction, and the relation `node_has_succ` relates it to its fall-through successor.

The state of the symbolic evaluator (line 14) is a record with a store mapping variables to values, and a path condition; an initial state (line 16) consists of an empty map and the `true` path condition.[1] ML-style functions are used to update and query the state. The function `update_store` (lines 18-19) updates a store binding, while the function `update_path_cond` (lines 21-24) adds another conjunct to the path condition, returning `none` in the case that the resulting path condition is unsatisfiable.

[1]We omit the definitions for maps; we use association lists with the standard operations `empty_map`, `get`, and `put`.

```
1  type val = i32 smt
2  type var = string
3  type operand = o_val(val) | o_var(var)
4  type binop = b_add | b_mul | b_eq | b_lt
5  type node = i32
6  type inst =
7    | i_binop(var, binop, operand, operand)
8    | i_jnz(operand, node) (* jump if the operand is not zero *)
9    | i_fail
10
11 input node_has_inst(node, inst)
12 input node_has_succ(node, node)
13
14 type state = { store: (var, val) map; path_cond: bool smt; }
15
16 fun initial_state : state = { store=empty_map; path_cond=`true`; }
17
18 fun update_store(x: var, v: val, st: state) : state =
19   { st with store=put(x, v, store(st)) }
20
21 fun update_path_cond(x: bool smt, st: state) : state option =
22   let y = path_cond(st) in
23   let z = `x /\ y` in
24   if is_sat(z) then some({ st with path_cond=z }) else none
25
26 fun operand_value(o: operand, st: state) : val * state =
27   match o with
28   | o_val(v) => (v, st)
29   | o_var(x) =>
30     match get(x, store(st)) with
31     | some(v) => (v, st)
32     | none => let v = `#{st}[i32]` in (v, update_store(x, v, st))
33     end
34   end
```

Fig. 2. A combination of types and input relations represent the program under evaluation; ML-style functions are defined for manipulating the complex types that represent evaluator state.

The built-in operator `is_sat` queries an external SMT solver for the satisfiability of its argument (an SMT proposition). The function `operand_value` (lines 26-34) looks up the value of an operand in the state, returning a pair of the value and a (possibly) new state. In the case that the operand is a value or a mapped variable, the relevant value is returned with the input state. In the case that the operand is a variable that is not in the store, the function returns a fresh symbolic bit vector (`` `#{st}[i32]` ``) with an updated state mapping the variable to that value.

The symbolic evaluator itself is defined through two IDB (i.e., `output`) relations (lines 36-37). The relation `reached` consists of tuples $(node, st, fuel)$ that indicate that the symbolic evaluator has

```
36 output reached(node, state, i32 option)
37 output failed(node, state)
38
39 fun decr(n: i32) : i32 option = if n > 0 then some(n - 1) else none
40
41 fun do_binop(b: binop, op1: operand, op2: operand, st: state) :
42     val * state =
43   let (v1, st1) = operand_value(op1, st) in
44   let (v2, st2) = operand_value(op2, st1) in
45   let fun b2i(x: bool smt) : i32 smt = `#if x then 1 else 0` in
46   let v = match b with
47           | b_add => `bv_add(v1, v2)`
48           | b_mul => `bv_mul(v1, v2)`
49           | b_eq => b2i(`v1 #= v2`)
50           | b_lt => b2i(`bv_slt(v1, v2)`)
51           end in
52   (v, st2)
53
54 reached(0, initial_state, some(10)). (* start with 10 units of fuel *)
55
56 reached(Next, St2, decr(N)) :-
57   reached(Curr, St, some(N)),
58   node_has_inst(Curr, i_binop(Def, B, Op1, Op2)),
59   node_has_succ(Curr, Next),
60   St2 =
61     let (v, st1) = do_binop(B, Op1, Op2, St) in
62     update_store(Def, v, st1).
63
64 reached(Dst, St2, decr(N)) :-
65   reached(Curr, St, some(N)),
66   node_has_inst(Curr, i_jnz(Op, Dst)),
67   some(St2) =
68     let (v, st1) = operand_value(Op, St) in
69     update_path_cond(`~(v #= 0)`, st1).
70
71 reached(Next, St2, decr(N)) :-
72   reached(Curr, St, some(N)),
73   node_has_inst(Curr, i_jnz(Op, _)),
74   node_has_succ(Curr, Next),
75   some(St2) =
76     let (v, st1) = operand_value(Op, St) in
77     update_path_cond(`v #= 0`, st1).
78
79 failed(Node, St) :-
80   reached(Node, St, _),
81   node_has_inst(Node, i_fail).
```

Fig. 3. Horn clauses and ML-style helper functions define the logic of the symbolic evaluator.

reached a node *node* with state *st* and the amount of fuel $fuel$. The relation `failed` consists of pairs (*node*, *st*) that indicate that the evaluator has reached a failure node *node* with the state *st*.

Before defining these relations, we define some helper functions. The function `decr` (line 39) decrements an integer if it is greater than zero, and else returns `none`; it is used to decrement the amount of fuel. The function `do_binop` (lines 41-52) is used to perform a binary operation on two operands. It looks up the value of those operands in the given state, and then returns a bit-vector-valued SMT formula representing the binary operation applied to those values. It also returns a state, since the resolution of the operands might have resulted in an updated state (if one of the operands is an unmapped variable). The locally scoped function `b2i` converts an SMT proposition to a bit-vector-valued SMT formula by building an if-then-else SMT expression (via the `#if · then · else` constructor) that is `1` if the proposition is true and `0` otherwise.

Four rules define the `reached` relation. The first one (line 54) states the base case: node `0` (the start of the CFG) is reachable with the initial state and 10 units of fuel. The remaining recursive rules match each possible step of execution and have a shared form: They check whether execution has reached a particular type of instruction with a non-zero amount of fuel, do whatever operation is required for that instruction, and then, if successful, step to the appropriate successor instruction with one less unit of fuel (computed via `decr`). For example, the second rule (lines 56-62) handles a binary operation: the operation is performed symbolically (via the function `do_binop`), the store is updated with the resulting value, and evaluation steps to the fall-through successor node `Next`.

The third and fourth rules define what happens when evaluation reaches a conditional jump. The first of these (lines 64-69) handles the case where the jump condition succeeds (i.e., when the operand in the jump can be nonzero) in which case the evaluator steps to the jump destination `Dst` with an updated path condition constraining the operand to be nonzero. The second of these (lines 71-77) handles the case where the jump condition fails (i.e, the operand can be zero). Note that these two cases are not mutually exclusive; the fact that these two rules can "fire" at the same time means that the symbolic evaluator can explore both branches in parallel.

One final rule (lines 79-81) defines the `failed` relation, and states that evaluation has uncovered a failure if it has reached a node with a fail instruction.

The symbolic evaluator can correctly determine that this program is safe:

```
if (x < y) { x++; assert(x <= y); }
```

It can also determine that this program is not (because the bit vector `y` can wrap around):

```
if (x < y) { x++; y++; assert(x <= y); }
```

While seemingly simple, this toy symbolic evaluator captures the essence of the more developed symbolic evaluator we describe as a case study (Section 5.4).

3.2 Overview

Formulog extends Datalog with a fragment of first-order ML and a language of SMT formulas (Figure 4). Accordingly, a program consists of Horn clauses, type and function definitions, and SMT declarations. The Horn clause fragment is the same as in Datalog, except with a richer variety of expressions *e* that can occur as arguments to predicates.

Type definitions. Formulog users can define ML-style algebraic data types, which can be polymorphic and mutually recursive. An algebraic data type definition consists of a list of type variables α, a type name D, and a list of constructors c with their argument types τ. Section 4 explains Formulog's type system in more detail; we provide a brief sketch now. Algebraic data types $D\ \tau^*$, base types B, and type variables α are treated as *pre-types*; intuitively, a pre-type t is the type of a concrete (non-formula) term. In addition to pre-types, there are types that represent SMT-relevant terms: a

Types

Types	τ	::=	$t \mid t\ \mathsf{smt} \mid t\ \mathsf{sym} \mid \mathsf{model}$
Pre-types	t	::=	$B \mid D\ \tau^* \mid \alpha$
Base types	B	::=	$\mathsf{bool} \mid \mathsf{string} \mid \mathsf{bv}[\mathsf{k}]_{k \in \mathbb{N}^+} \mid \ldots$

Terms

Programs	prog	::=	$H^*\ T^*\ F^*\ Z^*$
Horn clauses	H	::=	$p(e^*) :\!-\ P^*$
Premises	P	::=	$A \mid !A$
Atoms	A	::=	$p(e^*) \mid e = e$
Type definitions	T	::=	$\mathsf{type}\ \alpha^*\ D = \lceil c(\tau^*) \rceil^*$
Functions	F	::=	$\mathsf{fun}\ f(\lceil X : \tau \rceil^*) : \tau = e$
SMT declarations	Z	::=	$\mathsf{uninterpreted\ fun}\ c(\lceil t\ \mathsf{smt} \rceil^*) : t\ \mathsf{smt} \mid$ $\mathsf{uninterpreted\ sort}\ \alpha^*\ D$
Expressions	e	::=	$X \mid c(e^*) \mid k \mid f(e^*) \mid \mathsf{match}\ e\ \mathsf{with}\ \lceil c(X^*) \to e \rceil^* \mid$ $\mathsf{let}\ X = e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \otimes(e^*) \mid \grave{\phi}\grave{} \mid p(w^*)$
Constants	k	::=	$\mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \ldots$
SMT formulas	ϕ	::=	$,e \mid c^{\mathsf{SMT}}_{\mathsf{forall}}(\phi, \phi) \mid c^{\mathsf{SMT}}_{\mathsf{let}}(\phi, \phi, \phi) \mid c^{\mathsf{SMT}}_{\mathsf{ctor}}[c](\phi^*) \mid \ldots$
Wildcard	w	::=	$?? \mid e$
Values	$v \in \mathrm{Val}$	::=	$k \mid c(v^*)$

Namespaces

Data type names	D	$\in$	ADTVar	Variables	X	$\in$	Var
Type variables	α	$\in$	TVar	Predicates	p	$\in$	$\mathrm{PredVar}$
Constructors	c	$\in$	$\mathrm{CtorVar}$	Functions	f	$\in$	FunVar

Fig. 4. Formulog extends the abstract syntax of Datalog with type definitions, functions, SMT declarations, and a richer language of expressions.

t-valued SMT formula has type t smt, a t-valued SMT variable has type t sym, and an SMT model — a finite map from formula variables to concrete terms — has type model. The Formulog type system distinguishes the first three types where it is computationally relevant (i.e., during concrete evaluation, where confusing a t-valued formula for a concrete t term might lead to a computation getting stuck), and collapses them where it is not (i.e., during SMT evaluation, where there is no meaningful distinction between a t-valued formula and a concrete t value). It also prevents SMT models, which are not representable as SMT expressions, from flowing into SMT formulas.

Functions. Formulog supports ML-style function definitions, although functions are limited to being first-order and are not first-class values. They can be polymorphic and mutually recursive.

SMT declarations. Formulog users can declare uninterpreted functions and polymorphic uninterpreted sorts. An uninterpreted function amounts to a special constructor for building a purely symbolic term of type t smt (for some pre-type t). An uninterpreted sort amounts to a special symbolic pre-type t, where t is not inhabited by any value, but t sym and t smt are.

Expressions and formulas. Expressions e occur as function bodies and as predicate arguments in Horn clauses. Although Datalog traditionally limits ground terms to nullary constructors, we admit n-ary constructors. While this comes with the cost of possibly-diverging programs — adding

n-ary constructors makes Datalog Turing-complete [Green et al. 2013] — many recent Datalog variants allow complex terms, including Soufflé [Scholz et al. 2016], LogicBlox [Aref et al. 2015], and Flix [Madsen et al. 2016]. For us, complex terms provide a natural way to reify logical formulas, and they also can be used to create data structures that make it easier to encode certain analyses.

Additional Formulog expressions include standard ML fare like constants (booleans, strings, machine integers, and floats), function calls, and match, let, and if-then-else expressions. The expression $\otimes(e^*)$ represents the application of a primitive operator to a sequence of subexpressions. These cover both basic arithmetic operations (e.g., addition) and SMT-specific operations (e.g., checking for satisfiability, generating models; see Section 3.3.2).

The expression `` `$\phi$` `` is a quasi-quoted SMT formula, where the language of formulas ϕ consists of unquoted expressions $,e$ and formula constructors of the form $c^{\mathsf{SMT}}_{c'}$ applied to SMT formulas. Some of these constructors directly reflect SMT formula constructs; for example, the constructor $c^{\mathsf{SMT}}_{\mathsf{forall}}$ builds a universally quantified formula, and the constructor $c^{\mathsf{SMT}}_{\mathsf{let}}$ builds an SMT let formula. Formula constructors can appear only in formulas, and non-formula constructors cannot appear directly in formulas. We embed algebraic data type constructors in formulas using a family of *formula constructors*. Each formula constructor $c^{\mathsf{SMT}}_{\mathsf{ctor}}[c]$ lifts the user-defined algebraic data type constructor c to SMT. Quotes are used to delineate formulas and trigger a different type checking mode, in which the types t, t smt, and t sym are conflated (with some restrictions, as explained in Section 4). The unquote operator , escapes from this type checking mode and makes it possible to inject a non-formula expression into a formula. Section 3.3 discusses formulas in more detail.

We have already seen how the Datalog fragment of Formulog can include expressions from the ML fragment; the final expression $p(w^*)$ ties the loop by providing a way for the ML fragment to reference the Datalog fragment. The expression $p(w^*)$ acts like a function call that queries the contents of the relation p. Its exact behavior depends on its arguments, which are either expressions or the special wildcard term ??. If its arguments contain no wildcards, then $p(e^*)$ returns a boolean indicating whether the tuple identified by its arguments is in the p relation. If it has $k > 0$ wildcards, it returns a list of k-tuples: For each tuple v^* in the relation corresponding to p, there is a corresponding k-tuple in this list that is v^* projected to the wildcard positions; if there are n matching tuples in p, then the list is of length n. In other words, given complete arguments, a predicate is really just a predicate; given partial arguments with wildcards, a predicate is the multiset consisting of matching tuples after they have been appropriately projected.[2]

Remarks. Extending Datalog with our fragment of ML is not foundational, as it can relatively easily be translated to Datalog rules (this would not necessarily be the case for a higher-order fragment of ML). However, despite the fact that the ML fragment could be treated as just syntactic sugar, it has a significant positive impact on the usability of Formulog, as we argue in Section 6.

The concrete syntax of formulas in our prototype (and in the examples we give in this paper) differs from the abstract syntax given here. We differentiate between ML variables (initial lowercase) and Datalog variables (initial caps). Algebraic data type constructors are allowed to appear directly in formulas, and are implicitly lifted to the appropriate formula constructor (so data type constructor c is automatically lifted to $c^{\mathsf{SMT}}_{\mathsf{ctor}}[c]$). We do not support an explicit unquote operator; instead, we implicitly unquote variables, constants, and invocations of nullary functions. We support additional features (records, locally scoped functions, etc.) that can be easily compiled to the abstract syntax.

[2]Multisets can arise if the "anonymous" variable _ is used to project out unwanted columns. For example, given that exactly `p(1, 2)` and `p(3, 2)` hold, the expression `p(_, ??)` would evaluate to a multiset represented by the list `[2, 2]`.

Negation	`~`	:	`bool smt` → `bool smt`
Conjunction	`/\`	:	(`bool smt`, `bool smt`) → `bool smt`
Implication	`==>`	:	(`bool smt`, `bool smt`) → `bool smt`
Equality	`smt_eq[`t`]`	:	(t `smt`, t `smt`) → `bool smt`
SMT variable	`smt_var[`t'`,`t`]`	:	t' → t `sym`
Bit vector constant	`bv_const[`k`]`	:	`bv[32]` → `bv[`k`] smt`
Bit vector addition	`bv_add`	:	(`bv[`k`] smt`, `bv[`k`] smt`) → `bv[`k`] smt`

Fig. 5. Logical formulas are created in Formulog via built-in constructors, such as the ones shown here.

3.3 Logical Formulas

Formulog uses data types and operators to support constructing and reasoning about logical formulas. Formulog provides a library of data types that define logical terms. Most of the time during evaluation, these terms are unremarkable and treated just like any other ground term. However, these terms are interpreted as logical formulas when they are used as arguments to built-in operators that make calls to an external SMT solver. In our current prototype, it is possible to create logical terms in first-order logic extended with (fragments of) the SMT-LIB theories of uninterpreted functions, integers, bit vectors, floating point numbers, arrays, and algebraic data types [Barrett et al. 2016], as well as the theory of strings shared by the SMT solvers Z3 [de Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011].

3.3.1 Representing Formulas. Users create logical terms through constants and formula constructors. For example, to represent the formula $False \implies True$, one would use the term `` `false ==> true` ``, where `false` and `true` are the standard boolean values and `==>` is the infix constructor for implication.

Our current prototype offers around 70 constructors for creating logical terms ranging from symbolic string concatenation to logical quantifiers; others could be added in the future. Figure 5 shows a sample of these constructors and their types. Some constructors require explicit indices, either to guarantee that type information is available at runtime when the formula is serialized to SMT-LIB, or to make sure that the type of the arguments can be determined by the type of the constructed term (which makes type checking easier). For example, `bv_const[`k`]` creates a symbolic k-bit-vector value from a concrete 32-bit vector; at runtime, it is necessary to know the width k so that we can serialize it correctly. The constructor `smt_eq[`t`]` denotes the equality of two terms of type t `smt` (alternatively stated using the infix notation `#=`); here, the index makes sure that the type checker knows what types the arguments should have. A programmer typically does not need to provide these indices explicitly, as they can often be inferred (our prototype does this).

Formulog distinguishes between logic programming variables and formula variables. A formula variable is a ground term that, when interpreted logically, represents a symbolic value. A term `smt_var[`t'`,`t`]`(v) — typically abbreviated as `#{`v`}[`t`]` — is a formula variable of type t `sym` identified by a value v of type t'. Intuitively, v is the "name" of the variable. The term `#{`v`}[`t`]` is guaranteed not to occur in v, which means that the variable it represents is fresh with respect to the set of formula variables in v; this makes it easy to deterministically construct a new variable that is fresh with respect to an environment, a trick we use often in our case studies. For example, if `X` is bound to a list of boolean formula variables, the formula variable `#{X}[bool]` will not unify with any term in `X`. The shorthand `#`id`[`t`]` is equivalent to `#{"`id`"}[`t`]`, where id is a syntactically valid identifier.

Importantly, because formula variables are ground terms, we can derive facts containing formula variables without violating Datalog's range restriction, which requires that every derived fact is

Satisfiability	`is_sat`	:	`bool smt → bool`
	`is_sat_opt`	:	`(bool smt list, bv[32] option) → bool option`
Validity	`is_valid`	:	`bool smt → bool`
Model generation	`get_model`	:	`(bool smt list, bv[32] option) → model option`
Model inspection	`query_model`	:	`('a sym, model) → 'a option`

Fig. 6. Formulog provides built-in operators for reasoning about logical terms.

variable-free. This restriction enables efficient evaluation by simplifying table lookups, one of the fundamental operations in Datalog evaluation.

3.3.2 Using Formulas. Built-in operators provide a way to reason about logical terms as formulas (Figure 6). When an operator in the SMT interface is invoked, its formula argument is serialized into the SMT-LIB format and a call is made to an external SMT solver. These operators are assumed to act deterministically during a single Formulog run; an implementation can achieve this in the presence of a non-deterministic SMT solver by memoizing operations.

For example, to test the validity of the principle of explosion (any proposition follows from false premises), one could make the call `is_valid(` false ==> #x[bool] `)`. Like other operators, the SMT interface operators can be invoked from the bodies of rules, as here:

```
ok :-  #x[bool] != #y[bool],
       is_sat(`#x[bool] #= #y[bool]`) = true,
       is_sat(`~(#x[bool] #= #y[bool])`) = true.
```

This rule derives the fact `ok`: The term `#x[bool]` is not unifiable with the term `#y[bool]`, since they are different formulas, representing different SMT variables. But these terms both may and may not be equal when interpreted as formula variables via the operator `is_sat`. Within an invocation of `is_sat`, constraints are formed between `#x[bool]` and `#y[bool]` — in the first case they must be equal, and in the second case they must not be — but these constraints do not leak into the larger context. This is an intentional design decision and differs from the approach taken by paradigms like constraint logic programming (see Section 6).

Formulog provides two sets of operators for testing the satisfiability and logical validity of propositions. In general, an SMT solver can return three possible answers to such a query: "yes," "no," and "unknown." The operators `is_sat` and `is_valid` return booleans. In the case that the backend SMT solver is not be able to determine whether a formula ϕ is satisfiable, these operators fail (as explained in Section 4). The operator `is_sat_opt(`ϕ^*`, timeout)` provides more fine-grained control: it takes a list of propositions (interpreted as conjuncts) and an optional timeout, and returns an optional boolean, with `none` corresponding to "unknown." While we suspect that the simpler versions will be sufficient for most applications, this more complex version does allow applications to explicitly handle the "unknown" case if need be (e.g., pruning paths in symbolic execution).

The operator `get_model` takes a list of propositions and an optional timeout; it returns a model for the conjunction of the propositions if the SMT solver is able to find one in time, and `none` otherwise. The values of formula variables in this model can be inspected using `query_model`, which returns `none` if the variable does not occur free in the formula or if a concrete value for it is not representable in Formulog (for example, Formulog does not have a type for a concrete 13-bit vector). The values of symbolic expressions can be indirectly extracted through formula variables: Before finding the model, add the equality `` `$x$ #= $e$` `` to the formula, where x is a fresh formula variable and e is an expression; in the extracted satisfying model, x will be assigned the value of e in that model.

3.3.3 Custom Types in Formulas. Formulog's algebraic data types can be reflected in SMT formulas via SMT-LIB's support for algebraic data types. Thus, Formulog permits arbitrary term constructors to be used within logical formulas. For example, we can define a type `foo` with a single nullary constructor `bar` and then write formulas involving `foo`-valued terms:

```
type foo = | bar
ok :- is_valid(`#x[foo] #= bar`) = true.
```

This program would derive the fact `ok`: Since there is only one way to construct a `foo` — through the constructor `bar` — any symbolic value of type `foo` must be the term `bar`.

For each algebraic data type, we automatically generate two kinds of constructors that make it easier to write formulas involving terms of that type. The first kind is a constructor tester. For each constructor c of a type t, Formulog provides a constructor `#is_`c of type t `smt` $\rightarrow$ `bool smt`. The proposition `#is_`c`(`e`)` holds if the outermost constructor of e is c. The second kind is an argument getter. If c is a constructor for type t with n arguments of types t_i for $1 \leq 1 \leq n$, Formulog generates n argument getters of the form `#`c`_`i, where `#`c`_`i has the type t `smt` $\rightarrow$ t_i `smt`. When interpreted as a formula, the term `#`c`_`i`(`e`)` represents the value of the i^{th} argument of e. For example, we can state that a symbolic list of booleans is non-empty and its first argument is `true`:

```
`#is_cons(#x[bool list]) /\ #cons_1(#x[bool list])`
```

We could use the operator `get_model` to find a model of this satisfiable formula; in this model, `#x[bool list]` might be assigned the concrete value `cons(true, nil)`.

3.4 Operational Semantics

This section presents Formulog's operational semantics, making reference to a selection of the formal rules (Figures 7).[3,4] Formulog imposes the standard stratification requirements upon programs: no recursive dependencies involving negation or aggregation between relations. As a stratifiable program can be evaluated one stratum at a time, we focus on the evaluation of a single stratum.

A stratum is evaluated by repeatedly evaluating its Horn clauses until no new inferences can be made. The semantics of a Horn clause H is defined through the judgment $\vec{F}; \mathcal{W} \vdash H \rightarrow \mathcal{W}_\perp$, where a world $\mathcal{W}$ is a map from predicate symbols to sets of tuples (i.e., those that have been derived so far). A Horn clause takes a world to either a new world or the error value $\perp$. Going wrong can result for two reasons: either because a variable is unbound at a point where it needs to be bound, or because an operator is applied to a value outside of its domain. It is important to distinguish between a rule going wrong and a rule failing to complete because two terms fail to unify: The first is an undesirable error (ruled out by our type system), whereas the second is expected behavior.

A rule is evaluated by evaluating its premises one-by-one, using a left-to-right order (Clause). The judgment $\vec{F}; \mathcal{W}; \theta \vdash P \rightarrow \theta_\perp$ defines the semantics of a premise, which takes a world and a substitution θ (a partial function from variables to values) and returns a new substitution or an error. The substitution produced by one premise is used as the input to the next one. A successful inference extends the input world with a (potentially novel) tuple $\theta_n(\vec{X}_j)$, i.e., the result of element-wise applying the substitution produced by the rightmost premise to the variables in the head of the rule. Clause evaluation goes wrong if the evaluation of one of the premises goes wrong.

[3]Formulog can also be given a model-theoretic semantics: because the ML features can be desugared into Datalog rules, the model theory of Formulog is essentially that of stratified Datalog. The extended version of this paper [Bembenek et al. 2020] sketches this out further, and includes the full formalism for the operational semantics.

[4]In the boxed rule schemata, implicit parameters are in gray; we conserve space by stating the rules without threading implicit parameters through, which are unchanging. We write $\vec{x_i}$ for some metavariable x to mean a possibly empty sequence of xs indexed by i, and write $S_\perp$ for some set S to mean the set $S + \text{Err}$.

Namespaces and constructs

World	$\mathcal{W}$	$\in$	$\text{PredVar} \to \mathcal{P}(\text{Val}^*)$	Error	$\bot$	$\in$	Err
Substitution	θ	$\in$	$\text{Var} \rightharpoonup \text{Val}$	u-term	u	$::=$	$X \mid k \mid c(\vec{u_i})$

Clause semantics $\boxed{\vec{F}; \mathcal{W} \vdash H \to \mathcal{W}_\bot}$

$$\frac{|\vec{P_i}| = n \quad \theta_0 = \cdot \quad \forall i \in [0, n),\ \theta_i \vdash P_i \to \theta_{i+1}}{\vec{F}; \mathcal{W} \vdash p(\vec{X_j}) :\!- \vec{P_i} \to \mathcal{W}[p \mapsto \mathcal{W}(p) \cup \{\theta_n(\vec{X_j})\}]} \textsc{Clause}$$

Premise semantics $\boxed{\vec{F}; \mathcal{W}; \theta \vdash P \to \theta_\bot}$

$$\frac{\vec{v} \in \mathcal{W}(p) \qquad \theta \vdash \vec{X} \sim \vec{v} : \theta'_\bot}{\mathcal{W}; \theta \vdash p(\vec{X}) \to \theta'_\bot} \textsc{PosAtom} \qquad \frac{\theta \vdash Y \sim c(\vec{X}) : \theta'_\bot}{\mathcal{W}; \theta \vdash Y = c(\vec{X}) \to \theta'_\bot} \textsc{EqCtor}$$

Expression semantics $\boxed{\vec{F}; \mathcal{W}; \theta \vdash e \Downarrow_e v_\bot}$ $\boxed{\vec{F}; \mathcal{W}; \theta \vdash \vec{e} \Downarrow_{\vec{e}} \vec{v}_\bot}$

$$\frac{\mathcal{W}; \theta \vdash \vec{e} \Downarrow_{\vec{e}} \vec{v} \qquad [\![\otimes]\!](\vec{v}) = v}{\mathcal{W}; \theta \vdash \otimes(\vec{e}) \Downarrow_e v} \Downarrow_e\text{-Op} \qquad \frac{\mathcal{W}; \theta \vdash \phi \Downarrow_\phi v_\bot}{\mathcal{W}; \theta \vdash \text{`}\phi\text{`} \Downarrow_e v_\bot} \Downarrow_e\text{-Quote}$$

Formula semantics $\boxed{\vec{F}; \mathcal{W}; \theta \vdash \phi \Downarrow_\phi v_\bot}$ $\boxed{\vec{F}; \mathcal{W}; \theta \vdash \vec{\phi} \Downarrow_{\vec{\phi}} \vec{v}_\bot}$

$$\frac{\mathcal{W}; \theta \vdash \vec{\phi} \Downarrow_{\vec{\phi}} \vec{v}}{\mathcal{W}; \theta \vdash c^{\mathsf{SMT}}_{c'}(\vec{\phi}) \Downarrow_\phi c^{\mathsf{SMT}}_{c'}(\vec{v})} \Downarrow_\phi\text{-Ctor} \qquad \frac{\mathcal{W}; \theta \vdash e \Downarrow_e v}{\mathcal{W}; \theta \vdash ,e \Downarrow_\phi \text{toSMT}(v)} \Downarrow_\phi\text{-Unquote}$$

SMT conversion $\boxed{\text{toSMT}(v) = v}$

$$\text{toSMT}(c^{\mathsf{SMT}}_{\mathsf{let}}(v_1, v_2, v_3)) = c^{\mathsf{SMT}}_{\mathsf{let}}(v_1, v_2, v_3) \qquad \text{toSMT}(c^{\mathsf{SMT}}_{\mathsf{forall}}(v_1, v_2)) = c^{\mathsf{SMT}}_{\mathsf{forall}}(v_1, v_2)$$

$$\text{toSMT}(c(\vec{v_i})) = c^{\mathsf{SMT}}_{\mathsf{ctor}}[c](\overrightarrow{\text{toSMT}(v_i)}) \qquad \ldots$$

Fig. 7. A fragment of Formulog's operational semantics.

Without loss of generality, we assume that premises occur in a limited form: predicates are applied to only variables, written $p(\vec{X_i})$, and equality predicates bind variables, as in $Y = e$. (Our prototype similarly desugars premises.) An atom $p(\vec{X})$ is evaluated by non-deterministically choosing a tuple $\vec{v}$ from the tuples in $\mathcal{W}(p)$, and then pairwise unifying its elements with the variables $\vec{X}$ (PosAtom). The premise $Y = c(\vec{X})$ unifies its two terms (EqCtor). The judgment $\theta \vdash u_1 \sim u_2 : \theta_\bot$ defines the unification of terms u_1 and u_2 under the substitution θ; it results in an error if u_1 and u_2 both contain unbound variables, and a new substitution if they are otherwise unifiable.

Most expressions have standard semantics. An operator produces a value if its arguments are evaluated to values in its domain ($\Downarrow_e$-Op); it goes wrong if the argument values are outside its domain, e.g., if a string and number are added together. A quoted formula `$\phi$` evaluates to whatever ϕ evaluates to ($\Downarrow_e$-Quote). Formula $c^{\mathsf{SMT}}_{c'}(\vec{\phi})$ evaluates to formula $c^{\mathsf{SMT}}_{c'}(\vec{v})$ if arguments $\vec{\phi}$ evaluate to values $\vec{v}$ ($\Downarrow_\phi$-Ctor). If the expression e evaluates to the value v, then the formula ,e evaluates to the term $\text{toSMT}(v)$ ($\Downarrow_\phi$-Unquote), where the function toSMT lifts a term to its formula version.

4 TYPE SYSTEM

Formulog's type system is designed to meet three desiderata. The first desideratum is that concrete evaluation should never go wrong, which might happen if an operator is applied to an operand outside its domain or a variable is unbound at a point when it needs to be evaluated. The second desideratum is that SMT solving should never go wrong, which might happen if a term that does not represent a well-sorted formula under the SMT-LIB standard reaches the external SMT solver (e.g., a formula representing the addition of a 16-bit vector and 32-bit vector). The third desideratum is that the type system should make it easy to construct expressive logical formulas, including formulas that involve terms drawn from user-defined types.

There is some tension between the first and third of these desiderata. The first one requires that we differentiate between, for example, a concrete bit-vector value and a symbolic bit-vector value (e.g., a bit-vector-valued formula) since an operator that is expecting a concrete bit vector might get stuck if its argument is a symbolic bit vector. For instance, we want to rule out this program:

Example 1 (A bad program we would want to reject)**.**

```
type foo = | bar(bv[32])
fun f(x: foo) : bv[32] = match x with bar(y) => y + y end
not_ok :-  X = #x[bv[32]],
           f(bar(X)) = 42.
```

This program gets stuck evaluating `f(bar(X))`, since `y` is bound to a symbolic value in `f` but the ML fragment's addition operator needs concrete arguments. On the other hand, we are able to construct more expressive formulas if we can occasionally conflate concrete and symbolic expressions:

Example 2 (A good program we would want to accept)**.**

```
ok :-  X = #x[bv[32]],
       is_sat(`bar(X) #= bar(5)`) = true.
```

This rule asks whether there exists a symbolic bit vector x such that `bar(`x`)` equals `bar(5)`, where `bar` is the constructor defined above. This reasonable formula is not well-typed under a type system that uniformly distinguishes between concrete and symbolic values, since the constructor `bar` expects a concrete bit vector argument but instead receives the symbolic one x.

Formulog resolves the tension between these desiderata through a bimodal type system that acts differently inside and outside formulas (which are demarcated by quotations). In essence, the Formulog type system differentiates between the pre-type t, the SMT formula type t smt, and the SMT variable type t sym outside of formulas, but typically conflates them within formulas.[5] This bimodal approach disallows Example 1 (since outside a formula, a term of type bv[32] sym cannot be used where a term of type bv[32] is expected), while permitting Example 2 (since within a formula, a term of type bv[32] sym can be used anywhere a term of type bv[32] is expected).

Intuitively, this bimodal approach is safe because it distinguishes between concrete and symbolic values during concrete evaluation — where conflating them might lead to going wrong — and conflates them only during SMT evaluation, where the distinction is not meaningful. We have formalized the Formulog type system and proven it sound with respect to the operational semantics of Formulog. We present only a small subset of it here (Figure 8).[6]

The rule defining a well-typed Horn clause (H-Clause) depends on two notable judgments. The premise typing judgment $\Gamma \vdash P \rhd \Gamma'$ takes a variable typing context Γ and a premise P and

[5]It does not conflate them in binding positions where formula variables are required, such as in quantifiers.
[6]See the extended version of this paper [Bembenek et al. 2020] for the full formalism and proofs.

Contexts

$$\begin{array}{lrcl} \text{Data type declarations} & \Delta & ::= & \cdot \mid \Delta, D : \forall \vec{\alpha_i}.\ \{\overrightarrow{c_j : \tau_k}\} \\ \text{Program declarations} & \Phi & ::= & \cdot \mid \Phi, f : \forall \vec{\alpha}, \vec{\tau} \to \tau \mid \Phi, p \subseteq \vec{\tau} \\ \text{Variable contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha \end{array}$$

Clause typing $\boxed{\Delta; \Phi \vdash H}$

$$\frac{\cdot \vdash P_0 \rhd \Gamma_1 \qquad \ldots \qquad \Gamma_j \vdash P_j \rhd \Gamma_{j+1} \qquad \ldots \qquad \Gamma_n \vdash P_n \rhd \Gamma' \qquad p \subseteq \vec{\tau_i} \in \Phi \qquad \Gamma' \vdash \vec{X_i}, \vec{\tau_i} \rhd \Gamma'}{\vdash p(\vec{X_i}) :\!- \vec{P_j}} \; H\text{-Clause}$$

Variable binding and typing $\boxed{\Gamma \vdash x, \tau \rhd \Gamma}$ $\boxed{\Gamma \vdash \vec{x}, \vec{\tau} \rhd \Gamma}$

$$\frac{X \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash X, \tau \rhd \Gamma, X{:}\tau} \; X\tau\text{-Bind} \qquad \frac{\Gamma(X) = \tau}{\Gamma \vdash X, \tau \rhd \Gamma} \; X\tau\text{-Check}$$

Premise typing $\boxed{\Delta; \Phi; \Gamma \vdash P \rhd \Gamma}$

$$\frac{p \subseteq \vec{\tau_i} \in \Phi \qquad \Gamma \vdash \vec{X_i}, \vec{\tau_i} \rhd \Gamma'}{\Gamma \vdash p(\vec{X_i}) \rhd \Gamma'} \; P\text{-PosAtom} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash Y, \tau \rhd \Gamma'}{\Gamma \vdash Y = e \rhd \Gamma'} \; P\text{-Eq-FB}$$

Function and expression well formedness $\boxed{\Delta; \Phi \vdash F}$ $\boxed{\Delta; \Phi; \Gamma \vdash e : \tau}$

$$\frac{\mathrm{typeof}(\otimes) = \vec{\tau_i} \to \tau \qquad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \otimes(\vec{e_i}) : \tau} \; e\text{-Op} \qquad \frac{\Gamma \vdash \phi : \tau}{\Gamma \vdash \grave{}\phi\grave{} : \tau} \; e\text{-Quote}$$

SMT constructors and formula well formedness $\boxed{\Delta; \Phi; \Gamma \vdash c^{\mathsf{SMT}}_{\ldots} : \vec{\tau_i} \to \tau}$ $\boxed{\Delta; \Phi; \Gamma \vdash \phi : \tau}$

$$\frac{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{c}} : \vec{\tau_i} \to \tau \qquad \Gamma \vdash \phi_i : \tau_i}{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{c}}(\vec{\phi_i}) : \tau} \; \phi\text{-Ctor} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash ,e : \mathrm{toSMT}(\tau)} \; \phi\text{-Unquote} \qquad \frac{\Gamma \vdash \phi : t\ \mathsf{sym}}{\Gamma \vdash \phi : t\ \mathsf{smt}} \; \phi\text{-Promote}$$

SMT representations $\boxed{\mathrm{erase}(\tau) = t}$ $\boxed{\mathrm{toSMT}(\tau) = \tau}$

$$\begin{array}{rcl} \mathrm{erase}(B) & = & B \\ \mathrm{erase}(D\ \vec{\tau_i}) & = & D\ \overrightarrow{\mathrm{erase}(\tau_i)} \end{array} \qquad \begin{array}{rcl} \mathrm{erase}(t\ \mathsf{smt}) & = & \mathrm{erase}(t) \\ \mathrm{erase}(t\ \mathsf{sym}) & = & \mathrm{erase}(t) \end{array} \qquad \begin{array}{rcl} \mathrm{toSMT}(t) & = & \mathrm{erase}(t)\ \mathsf{smt} \\ \mathrm{toSMT}(t\ \mathsf{smt}) & = & \mathrm{erase}(t)\ \mathsf{smt} \\ \mathrm{toSMT}(t\ \mathsf{sym}) & = & \mathrm{erase}(t)\ \mathsf{sym} \end{array}$$

Fig. 8. A fragment of Formulog's type system.

produces a new variable typing context Γ'. The variable binding and typing judgment $\Gamma \vdash x, \tau \rhd \Gamma'$ holds if either X is not in Γ, in which case Γ' extends Γ with X mapped to τ ($X\tau$-Bind), or X is mapped to τ by Γ, in which case $\Gamma = \Gamma'$ ($X\tau$-Check). As can be seen from rule H-clause, the type checking of Horn clauses is flow-sensitive and proceeds left-to-right across the clause, with the "output" context of checking premise P_i used as the "input" context for checking premise P_{i+1}.

This left-to-right type checking mirrors the left-to-right evaluation strategy Formulog uses; this is important for ensuring that variables are bound at the correct points.[7] The second line of premises in rule H-CLAUSE ensures that every variable in the head of the rule is bound at the type specified by the head relation's signature.

A positive atom is well typed if each of its variable arguments has the type given to that argument by the relation's signature (P-POSATOM). A premise of the form $Y = e$ is typed according to a few different rules depending on which side of the equation is ground with respect to the input context Γ. The key is that our type system only types premises of the form $Y = e$ when unification is guaranteed to not go wrong at runtime.

The typing rules for most expressions are standard. An operation is well-typed if its arguments match its type signature (ϕ-CTOR). A quoted formula `` `$\phi$` `` types at whatever ϕ types at (e-QUOTE). The formula constructor $c_{\mathsf{c}}^{\mathsf{SMT}}$ is well typed if the types of its arguments match its type signature (ϕ-CTOR); in the case of a constructor for an algebraic data type that has been lifted to a formula constructor, that signature will require the constructed term and all of its arguments to have types of the form t smt. If an expression types at τ, then the formula ,e types at $\mathrm{toSMT}(\tau)$ (ϕ-UNQUOTE). The helper function toSMT lifts a type to a formula type; for example, it lifts bool to bool smt. The typing rules for formulas also include a rule promoting from t sym to t smt, reflecting the fact that, within a formula, a t-valued formula variable can be used anywhere a t-valued formula can be.[8]

Type soundness with respect to the semantics of Formulog comes from safety and preservation:

THEOREM 4.1 (SAFETY). *If* $\Delta; \Phi \vdash \vec{F}_i\ \vec{H}_j$ *and* $\Delta; \Phi \models \mathcal{W}$ *then for all* $H \in \vec{H}_j$, $\neg(\vec{F}_i; \mathcal{W} \vdash H \to \bot)$.

THEOREM 4.2 (PRESERVATION). *If* $\Delta; \Phi \vdash \vec{F}_i\ \vec{H}_j$ *and* $\Delta; \Phi \models \mathcal{W}$ *and* $\vec{F}_i; \mathcal{W} \vdash H \to \mathcal{W}'$ *for some* $H \in \vec{H}_j$ *then* $\Delta; \Phi \models \mathcal{W}'$.

Safety (Theorem 4.1) guarantees that a Horn clause from a well-typed program, evaluated on a well-typed world (i.e., one where all the tuples have the right types), cannot step to error. Thus, safety means that an operator is never applied to an operand outside its domain, and a variable is never unbound when it needs to be bound. Preservation (Theorem 4.1) guarantees that if a Horn clause, from a well-typed program, is evaluated on a well-typed world and results in a new world, then that new world is also well-typed. Taken together, these theorems imply that a well-typed Formulog program does not go wrong during concrete evaluation.

The type system is sound with respect to the semantics of SMT-LIB because the types of the formula constructors provided by Formulog are consistent with the types given by the SMT-LIB standard. The Formulog type system guarantees that, at runtime, terms (including formulas) are well-typed, and the type system prevents terms that are not representable in SMT (such as those of type model) from flowing into SMT formulas. We distinguish between SMT-compatible types and non-SMT types formally by indexing the type well formedness judgment with a *mode*, which is either smt (for those types that can be sent to the solver) or exp (for those types that cannot). It is fair to think of these modes as kinds with a subkinding relationship: types of kind smt can safely be treated as general types of kind exp, but not the other way round.

Assumptions. An actual implementation of Formulog, such as our prototype, has to contend with a few sources of going wrong that are not captured in our formal model. First, our model assumes that patterns in match clauses are exhaustive; this is just for simplicity, and could be statically

[7] The fact that the operational semantics and type system assume a certain order of evaluation does not prohibit a Formulog runtime from reordering premises within rules (for example, when applying database-style query planning optimizations); it just needs to check that the new order is also well typed. This type of rewriting does not affect the result of running the rule provided that all subexpressions terminate (an assumption we make).

[8] The opposite is not true, since some formula constructors (i.e., quantifiers and let expressions) bind formula variables.

checked using standard algorithms. Second, our model assumes that operators are total with respect to terms with the correct type. There are three places where this assumption might break: 1) division or remainder by zero; 2) the operators `is_sat` or `is_valid` may induce an "unknown" response from the external SMT solver; and 3) the SMT solver may reject patterns used in trigger-based quantifier instantiation that it considers to be ill formed (for example, if the pattern contains a binding operation). The first case is standard for many languages; the second can be avoided if the programmer uses the option-returning SMT operator `is_sat_opt`. The last case would be hard to check statically; however, an implementation could dynamically check patterns before making a call to the SMT solver, dropping invalid patterns and issuing a warning to the user. Our prototype uses "hard exceptions" by default, aborting the program. We also support a "soft exception" mode, which treats all these cases analogously to unification failures, halting execution on the current path but allowing execution on other paths to continue.

5 IMPLEMENTATION AND CASE STUDIES

In this section we briefly describe our prototype implementation of Formulog, and then discuss three analyses we have built as case studies: refinement type checking, bottom-up points-to analysis, and bounded symbolic evaluation.

5.1 Prototype

Our prototype runtime (~17.5K LOC Java) works in five stages: parsing, type checking, rewriting (for query specialization), validation, and evaluation. Stratification and the range restriction are checked during the validation phase. Our parallel implementation of semi-naive evaluation [Bancilhon 1986] uses a work-stealing thread pool; worker threads dispatch SMT queries to external solvers (Z3 by default [de Moura and Bjørner 2008]). Our prototype is feature complete, but not very optimized.[9]

Unless otherwise noted, we ran experiments on an Ubuntu Server 16.04 LTS machine with a 3.1 GHz Intel Xeon Platinum 8175 processor (24 physical CPUs, each hyperthreaded) and 192 GiB of memory. We configured our Formulog runtime to use up to 40 threads and up to 40 Z3 instances (v4.8.7); all comparison systems were set to use the same version of Z3 (with one exception, noted later). For each result, we report the median of three trials.[10] Times are given as minutes:seconds.

5.2 Refinement Type Checking

We have implemented a type checker in Formulog for Dminor, a first-order functional programming language for data processing that combines refinement types with dynamic type tests [Bierman et al. 2012]. This type system can, e.g., prove that

```
x in Int ? x : (x ? 1 : 0)
```

type checks as `Int` in a context in which `x` has the union type `(Int|Bool)`. Proving this entails encoding types and expressions as logical formulas and invoking an SMT solver over these formulas. We built a type checker for Dminor by almost directly translating the formal inference rules used to describe the bidirectional Dminor type system. In fact, we programmed so closely to the formalism that debugging an infinite loop in our implementation helped us, along with the Dminor authors, uncover a subtle typo in the formal presentation! Our Dminor type checker is 1.2K lines of Formulog. The implementation of Bierman et al. is 3.2K lines of F♯ and 400 lines of SMT-LIB; we estimate that the functionality we implemented accounts for over two thousand of these lines.[11]

[9] Our prototype is available at https://github.com/HarvardPL/formulog.
[10] For each case study, we use a tool to translate the input programs into Formulog facts. We do not include these times, which are typically quite short. Extracting libraries can take a few minutes, but this needs to be done only once per library.
[11] The reference implementation is closed source; the authors have kindly provided us with line counts for each file.

```
fun accum_nil_axiom : bool smt =
  let (f, i) = (#func[closure], #init[enc_val]) in
  `forall f, i : accum(f, v_zero, i). accum(f, v_zero, i) #= i`
```

Fig. 9. This axiom encodes the denotation of a Dminor accumulate expression over an empty multiset. The term in the formula between : and . is a quantifier pattern [Detlefs et al. 2005].

```
fun encode_type(t: typ, v: enc_val smt) : bool smt * bool smt =
  match t with
  | t_any => (`true`, `true`)
  | t_bool => (`#is_ev_bool(v)`, `true`)
  | t_coll(s) =>
    let x = #{(s, v)}[enc_val] in
    let (phi, ax) = encode_type(s, `x`) in
    (`good_c(v) /\ forall x : mem(x, v). mem(x, v) ==> phi`, ax)
```

Fig. 10. This function (fragment) constructs a formula capturing the logical denotation of a Dminor type.

The encoding of Dminor types and expressions is complex, requiring uninterpreted sorts, uninterpreted functions, universally quantified axioms, and arrays (among other features). The fact that we were able to code this relatively concisely speaks to the expressiveness of Formulog's formula language. For example, Figure 9 shows an axiom describing the denotation of the base case of a Dminor accumulate expression, which is essentially a fold over a multiset. Here, the type `closure` is an uninterpreted sort, `enc_val` is an algebraic data type that represents an encoded Dminor value, and `accum` and `v_zero` are uninterpreted functions, where the latter represents an empty multiset.

We defined a set of mutually-recursive functions that encode expressions, environments, and types. For example, the type encoding function (fragment, Figure 10) takes a type τ and an (encoded) Dminor value v, and returns two propositions. The first is true when v has type τ. The second is a conjunction of axioms: new axioms are created to describe the denotation of the bodies of accumulate expressions as they are encountered when encoding expressions. The first case in the figure encodes the fact that any value has type `Any`. The second one says that a value has type `Bool` if it is constructed using the constructor `ev_bool`; the constructor `#is_ev_bool` is an automatically-generated constructor tester. The third case handles multiset types. It creates a fresh encoded value `x`, uses `x` to recursively create a proposition representing the encoding of the type `s` of items in the multiset, and then returns a proposition requiring the value to be a "good" collection (defined using the uninterpreted function `good_c`) and every item in the multiset to have type `s` (where `mem` is another uninterpreted function).

Although we use ML-style functions to define the logical denotation of expressions, environments, and types, we use logic programming rules to define the bidirectional type checker, which allows us to write rules that are very similar to the inference rules given in the paper. Figure 11 gives the one rule defining the subtype relation: `T` is a subtype of `T1` in environment `Env` if `T1` is well formed and the denotation of `T`, given our axioms and the denotation of `Env`, implies the denotation of `T1`. This rule is an almost exact translation of the inference rule given in the paper.

Finally, the type checker needs to ensure that any expressions that occur in refinements are pure (i.e., terminate and are deterministic). We have written a termination checker based on the size-change principle [Lee et al. 2001]. Our implementation is another good example of the synergy

```
subtype(Env, T, T1) :-
  type_wf(Env, T1),
  encode_env(Env) = Phi_env,
  X = `#{(Env, T, T1)}[enc_val]`,
  encode_type(T, X) = (Phi_t, Axioms1),
  encode_type(T1, X) = (Phi_t1, Axioms2),
  Premises = [Phi_t, Phi_env, Axioms2, Axioms1, axiomatization],
  is_sat_opt(`~Phi_t1` :: Premises, z3_timeout) = some(false).
```

Fig. 11. This rule defines Dminor's semantic subtyping relation. It uses the operator `is_sat_opt` instead of `is_valid` because its SMT queries can sometimes result in "unknown."

between ML-style functions and Datalog rules, as we use the former to define the composition of two size-change graphs and use the latter to find the fixed point of composing size-change graphs.

We tested our type checker on six of the sample programs included in the Dminor documentation (the other three examples make use of a feature — the ability to generate an instance of a type — that we did not implement, although it should be possible to do so; to the best of our knowledge, these are the only publicly available Dminor programs). We combined these examples into a single aggregate program of ∼150 LOC. The reference implementation type checked this program in 1.5 seconds using an optimization that tries syntactic subtyping before semantic subtyping; with this optimization disabled, it took 3.6 seconds.[12] Our implementation completed in 4.7 seconds; it did not use this optimization (which is not detailed in the paper), but did use a newer version of Z3. Thanks to parallelization, our implementation automatically scaled to larger programs: On a synthetic program consisting of ten copies of the original aggregate program, it completed in 19.8 seconds (2.0 seconds per program copy); on a synthetic program consisting of 100 copies, it completed in 153.6 seconds (1.5 seconds per program copy). In contrast, the reference implementation did not scale: even with the syntactic-subtyping optimization enabled, it took 68 seconds on the ten-copy program and over 100 minutes on the 100-copy program.

5.3 Bottom-up Points-to Analysis

We have implemented the bottom-up context-sensitive points-to analysis for Java proposed by Feng et al. [2015]. A points-to analysis computes a static approximation of the objects that stack variables and heap locations can point to at runtime. A bottom-up points-to analysis does this through constructing method summaries that describe the effect of a method on the heap; it is bottom-up in the sense that summaries are propagated up the call graph, from callees to callers.

In Feng et al.'s algorithm, a method summary is an abstract heap that maps abstract locations to heap objects, where an abstract location might be a stack variable, an explicitly allocated heap object, or an argument-derived heap location. Edges in the abstract heap are labeled with logical formulas that describe the conditions under which the edges hold; when a method summary is instantiated at a call site, a constraint solver can be used to filter out edges with unsatisfiable labels.

Feng et al.'s tool based on this algorithm, Scuba, is ∼15K lines of Java, builds on the Chord analysis framework [Naik 2011], and uses Z3 to discharge constraints. As for many realistic static analysis tools, there is a gap between what is implemented in Scuba and the formal specification of the analysis. This is partly because Scuba is written in Java: Object-oriented programming does not naturally capture inference rules, the form of the specification. In contrast, our Formulog

[12]Here we used a machine with Microsoft Windows Server 2019 and the same hardware specs as our Ubuntu machine.

```
instantiate_ptsto(C, O1, Phi1, O2, widen(C, Phi_all)) :-
  instantiate_loc(C, heap(O1), heap(O2), Phi2),
  instantiate_constraint(C, Phi1, Phi3),
  Phi_all = conjoin(Phi2, Phi3).
```

Fig. 12. This rule describes how a points-to edge to object `O1` labeled with constraint `Phi1` is instantiated at a call site `C`: if at `C` a heap location `heap(O1)` can be instantiated to a heap location `heap(O2)` under constraint `Phi2`, and the original constraint on the edge `Phi1` can be instantiated to a constraint `Phi3`, then the points-to edge to `O1` labeled with `Phi1` instantiates to a points-to edge to `O2` labeled with `widen(C, Phi_all)`, where `Phi_all` is the conjunction of `Phi2` and `Phi3` and `widen` is a function that widens constraints in mutually-recursive functions (one of the heuristics we borrowed from Scuba).

Table 1. In the median, our implementation of a bottom-up points-to analysis for Java was 6.7× slower than Scuba, the reference implementation (times in mm:ss); however, the two tools use different heuristics and thus compute very different things, as indicated by the discrepancy in the number of points-to edges computed in the summary for `main` (which also captures the effect on the heap of methods invoked transitively from it).

	Scuba		Formulog	
Benchmark	Time	# `main` edges	Time	# `main` edges
antlr	1:11	3,313	12:16	112,415
avrora	1:05	714	7:40	127,535
hedc	0:57	867	5:04	2,962
hsqldb	0:51	780	4:53	7,039
luindex	1:40	3,395	T/O	-
polyglot	0:55	117	4:52	4,245
sunflow	3:48	7,456	T/O	-
toba-s	0:58	521	4:57	12,284
weblech	1:10	1,262	17:58	6,785
xalan	0:54	183	5:40	55,722

implementation, which is ∼1.5K LOC, closely mirrors the inference rules. For example, we can directly state how a points-to edge is instantiated at a call site (Figure 12), one step of summary instantiation, a complex process defined through half a dozen mutually recursive relations that need to be computed as a fixed point. The Java code for encoding this logic is more complex and further from the formal specification. Programming close to the specification also helps check the specifications' correctness: while implementing in Formulog one of the judgments specified by Feng et al., we discovered an inconsistency between the judgment's definition and its type signature.

Scuba employs a range of sophisticated heuristics that are essential to making the algorithm perform in practice, as they tune precision to achieve scalability. Some go far beyond the algorithm described in the paper and are interesting in their own right. Our implementation uses some heuristics based on the ones in Scuba. The fact that we were able to implement useful heuristics — a necessity for a realistic static analysis tool — argues for the practicality of Formulog. Moreover, we were able to do so such that our code still closely reflects the core algorithm specified in the paper.

We ran both tools on the benchmarks used in the evaluation by Feng et al., which represent a selection from the pjbench suite plus the benchmark polyglot.[13] These experiments include library code and use a context-sensitivity of two call sites; reflection is ignored, as are many native methods.

[13]The pjbench suite is available at https://bitbucket.org/psl-lab/pjbench/src/master/.

Given an hour timeout, our implementation completed on eight of the ten benchmarks, with times ranging from five to 18 minutes (Table 1). In the median, we were 6.7× slower than Scuba. However, a performance comparison between the tools should be taken with a grain of salt: Since they use different heuristics, they compute very different things.

In sum, we were able to implement the algorithm in a way that is still very close to its specification and achieve decent performance on many realistic benchmarks while implementing only a small selection of heuristics. Other heuristics might have helped our version complete on the two benchmarks it timed out on. Making the algorithm practical is a significant engineering challenge: Even with its sophisticated heuristics, Scuba does not complete on all benchmarks in pjbench.[14]

Moreover, our implementation could be used as a platform for exploring potential optimizations to Scuba. First, because it is automatically parallelized (with a user-chosen number of worker threads), it could be used to evaluate how well the underlying points-to algorithm parallelizes before going through all the trouble of parallelizing Scuba, which uses mutable state in a complex way. Second, thanks to the magic set transformation, we have automatically derived a goal-directed version of the analysis that computes only the summaries necessary for constructing user-requested summaries. The points-to algorithm resulting from this transformation could be used as a road map for implementing a demand-driven version of Scuba, which Feng et al. describe as future work.

5.4 Bounded Symbolic Evaluation

We have written a symbolic evaluator (∼1K LOC) for a fragment of LLVM bitcode [Lattner and Adve 2004] corresponding to a simple imperative language with integer arrays and symbolic integers (a symbolic integer represents a set of integer values that might occur at runtime). It implements a form of bounded symbolic execution [King 1976], exploring all feasible program paths up to a given length, evaluating concretely whenever possible, and aggressively pruning infeasible paths.

Our implementation uses a different logic rule to define each of the possible cases during evaluation, and uses ML functions to manipulate and reason about complex terms representing evaluator state. For example, one rule defines when an assertion fails (Figure 13). This rule says that the path `Path` ends in a failure with evaluator state `St` if: (1) there is an assert instruction `Instr` with argument `X`, (2) following `Path` has led the evaluator to that instruction with state `St`, (3) `X` could have the (possibly symbolic) integer value `V` in state `St`, and (4) `V` may be zero. The function `may_be_zero(V, St)` returns true if and only if `V` may be zero given `St`. We represent symbolic values as SMT formulas, so when `V` is symbolic, this function invokes the SMT solver.

We have evaluated our symbolic evaluator on ten benchmarks based on five template programs. The first template (shuffle-N) non-deterministically shuffles an array of size N and asserts that the resulting array represents the same set as the input array. The second template (sort-N; Figure 14) splits into two branches, sorts an array using selection sort in both branches, and asserts that the resulting array is sorted in the second branch. The third template completes a partially filled-in 4×4 grid of integers, such that there is a path from 1 to 16 where each integer follows its predecessor and only horizontal and vertical movements are used; the benchmark numbrix-sat runs this program on a satisfiable instance, while the benchmark numbrix-unsat runs it on an unsatisfiable one. The fourth template (prioqueue-N) tests the equivalence of two implementations of a priority queue (one based on a heap, the other on an unsorted array) by pushing the same N symbolic integers on them and verifying that they have the same behavior during a sequence of operations. The fifth template (interp-N) runs an interpreter for a simple bytecode language for N steps; the input bytecode is represented by an array of symbolic integers that can be interpreted as commands for binary operations, register loads and stores, and conditional jumps.

[14]For example, we found it timed out on batik, chart, fop, lusearch, and pmd (as did our implementation).

```
failed_assert(Path, St) :-
  assert_instruction(Instr, X),
  stepped(Instr, St, _, Path),
  has_value(X, St, v_int(V)),
  may_be_zero(V, St) = true.
```

Fig. 13. This rule states that the symbolic evaluator has reached a failing assertion when the argument of the assert instruction may be zero.

```
a := array of N symbolic ints;
b := symbolic int;
if (b) { sort a; }
else { sort a; assert a sorted; }
```

Fig. 14. This pseudocode sketches a C program that creates an array, branches, sorts it in each branch, but asserts that the result is sorted only in one branch.

Table 2. We report absolute times (mm:ss) for KLEE, CBMC, and a Formulog-based symbolic evaluation tool on ten benchmark programs; for the latter, we also report speedups (↑) and slowdowns (↓) relative to KLEE.

Benchmark	# paths	KLEE	CBMC	Formulog
shuffle-4	125	0:06	0:01	0:02 (↑ 3.0×)
shuffle-5	1,296	1:56	0:01	0:07 (↑ 16.6×)
sort-6	2,718	2:29	0:24	0:18 (↑ 8.3×)
sort-7	22,070	27:13	2:46	3:16 (↑ 8.3×)
numbrix-sat	1	0:15	0:01	1:10 (↓ 4.7×)
numbrix-unsat	1	0:15	0:01	0:59 (↓ 3.9×)
prioqueue-5	1,132	0:43	6:45	0:16 (↑ 2.7×)
prioqueue-6	4,409	3:24	T/O	1:10 (↑ 2.9×)
interp-5	994	0:55	0:05	0:39 (↑ 1.4×)
interp-6	3,433	3:19	0:12	T/O (↓ ∞×)

We compared our times on these benchmarks against the symbolic execution tool KLEE (v2.1) [Cadar et al. 2008] and the bounded model checker CBMC (v5.11) [Clarke et al. 2004] (Table 2); we used a timeout of 30 minutes. These should not be taken as apples-to-apples comparisons: KLEE operates over all of LLVM bitcode and CBMC operates over C source code, whereas we handle just a fragment of LLVM bitcode; CBMC implements bounded model checking and not symbolic execution, with the result that it generates many fewer (but presumably more complex) SMT queries; and all three tools might translate program constructs into SMT formulas in different ways, leading to different external solver performance. Nonetheless, these comparisons provide some context for our evaluation numbers.

In general, our tool achieved speedups over KLEE, but did not quite match the performance of CBMC. It performed relatively poorly for benchmarks with a single path (numbrix-sat and numbrix-unsat), but on most other programs we were able to achieve substantial speedups (1.4×-16.6×) over KLEE and perform within striking distance of CBMC. This was at least partly due to the fact that our analysis is automatically parallelized, whereas KLEE and CBMC are single threaded. The interp-N benchmarks caused trouble for our tool: Our trials for interp-5 had an unusually high degree of variance (with two trials taking less than 40 seconds, and one trial taking ~18 minutes), and our tool timed out on interp-6. We suspect that this might be because, on this benchmark, our tool generates SMT queries involving the theory of arrays, and our particular naive encoding might be leading to slowdowns with the external SMT solver.[15]

[15]The shuffle-N benchmarks are the only other ones during which our tool generates SMT queries with array constructs.

Table 3. Formulog analyses can be concise and close to the formal specifications. This table gives the number of rules, non-nullary functions, and line counts for our case studies; in parentheses, we give the number of rules and functions that correspond to the formal specifications (the rest handle other parts of the analyses, e.g., the termination checker in Dminor, and the context-insensitive points-to analysis used by the bottom-up points-to analysis). For comparison, we provide the number of rules and functions used in the formal specifications, as well as the line count of the reference implementations. Our symbolic evaluator is not based on a particular specification; we omit a LOC comparison with the reference implementations, KLEE and CBMC, as they handle much larger input languages and it would be difficult to isolate the parts of their codebases that correspond to the language our analysis supports.

	Formulog impl.			Specification		Reference impl.
Analysis	# rules	# funcs	LOC	# rules	# funcs	LOC
Dminor type checker	78 (50)	61 (43)	1.2K	34	15	~2K lines F♯ & SMT-LIB
Bottom-up points-to	203 (47)	49 (28)	1.5K	19	8	15K lines Java
Symbolic evaluator	51	37	1K			

Additionally, our tool can be run in a goal-directed mode: If we only want to check that no assertion fails, we can add the query `failed_assert(_Path, _St)`, triggering the Formulog runtime to rewrite our evaluator to explore only paths that could potentially lead to a failed assertion. For the sorting benchmarks (Figure 14), this means the symbolic evaluator can ignore the first branch of the program. This leads to significant performance gains, as we completed sort-6 in 13 seconds and sort-7 in 2:18, representing increased speedups of 11.5× and 11.8×, respectively, over KLEE. We ran CBMC in a similar directed mode (it can use program slicing [Weiser 1984] to ignore parts of the program irrelevant to assertions); it was slightly slower than our Formulog implementation, completing sort-6 in 28 seconds and sort-7 in 2:25. This suggests the potential of Formulog's automatic optimizations, which help make it competitive with hand-optimized systems.

6 DESIGN EVALUATION

In this section, we evaluate the design of Formulog with respect to our case studies. We argue that Formulog is an effective and usable tool for writing SMT-based analyses.

Formulog makes it possible to write SMT-based analyses in a way that is close to their mathematical specification, leading to concise encodings (Table 3). Our implementations of the Dminor type checker (Section 5.2) and the bottom-up points-to analysis (Section 5.3) directly mirror their published formal specifications; our third case study (Section 5.4), which was not based on any particular formalization, would itself be the basis of a reasonable specification of symbolic evaluation. Formulog provides language features that are a good match for the way that SMT-based analyses are specified: algebraic data types naturally encode BNF grammars (a common feature in analysis specifications); Horn clauses match judgments; ML functions fit helper functions; and the reification of formulas as terms captures the way that formulas are treated in analysis specifications.

As a corollary, analyses written in Formulog can be concise. Despite encoding quite complex logic, each of our case studies is less than 1.5K lines of code. In the case of the points-to analysis, this is 10× smaller than the reference implementation (which also uses functionality defined externally in Chord). This is partly because Scuba implements heuristics that we do not and Java is a verbose language; however, we suggest that much of the difference is because Formulog is a better fit for encoding the logic of the analysis than an imperative, object-oriented language like Java. The relative concision of Formulog matches the results reported by previous work on Datalog-based static analysis, which found that Datalog-based analyses can be orders of magnitude more concise

than counterparts written in more traditional languages [Whaley et al. 2005]. The ML fragment of Formulog also helps it be concise, since ML expressions — through supporting sequenced, nested, and scoped computation — can encode logic that would be more verbose to write in Datalog.

We have shown that three diverse case studies can be naturally encoded in Formulog, suggesting that its design is a good match for a range of SMT-based analyses. However, not all analysis logic can be easily encoded in Formulog. There is currently no way to join facts, a useful operation for abstract interpretation-based analyses [Cousot and Cousot 1977]. The restriction to stratified negation is sometimes too severe: For example, one Dminor rule for the type synthesis relation `synth` is not directly expressible in Formulog, because it is defined in terms of the negation of the type well formedness relation, which is in turn defined using the relation `synth`.[16] Finally, given its lack of mutable state, Formulog is probably not a good fit for analyses that can most naturally be specified in an imperative manner, such as lazy abstraction model checking [Henzinger et al. 2002].

Because Formulog is designed to be compatible with Datalog, we can expand the type of analysis logic it supports by taking advantage of research on Datalog extensions. For instance, lattice-based recursive aggregation [Madsen et al. 2016; Szabó et al. 2018] would make it possible to join facts, and local stratification [Przymusinski 1988] would support the Dminor logic we previously cited.

Formulog provides a rich and flexible language of formulas that supports the type of logic-based reasoning found in SMT-based analyses. The formula fragment of Formulog makes it possible to use formulas the way they need to be used by static analyses. A good example of this is the decision to reify logical formulas as terms, a departure from the approach taken by constraint logic programming [Jaffar and Lassez 1987; Jaffar and Maher 1994] and constrained Horn clause (CHC) solving [Bjørner et al. 2015; Grebenshchikov et al. 2012; Gurfinkel et al. 2015; Hoder and Bjørner 2012], the two major previous paradigms for combining logic programming and constraint solving. In these systems, constraints are represented as predicates, not terms, and an inference is made if the constraints in the body of a rule are satisfiable. This approach makes sense in the context of programming with *constraints*; however, it seems overly restrictive in the context of programming with *formulas*, which do not necessarily have to be used directly as constraints. For example, analyses like our Dminor type checker need to check the validity of a formula, which is the unsatisfiability of its negation. Checking validity does not easily fit in constraint-based paradigms, since constraint programming is built around satisfiability. Similarly, we might want to write an analysis that uses Craig interpolants [Craig 1957]. One could imagine extending Formulog's SMT interface to include an operator `interpolate` that takes two formulas and returns a third (optional) formula, the interpolant; it is not clear how to do this in one of the constraint-based paradigms.

Our treatment of formula variables through the constructor $\#\{e\}[t]$ provides further evidence. This mechanism makes it easy to identify a formula variable with an object-level construct (e.g., a variable in the input program) by choosing for e the expression representing that construct. It also makes it easy to create a variable that is guaranteed to be fresh relative to a set of constructs (e.g., fresh with respect to an environment), an extremely useful operation. This is done by choosing for e a tuple of the constructs that the variable needs to be fresh with respect to. We use this trick in both the Dminor type checker and the symbolic evaluator. Crucially, this freshness mechanism is deterministic, which means that we can safely rewrite Formulog programs and parallelize them. The logic programming language Calypso [Aiken et al. 2007; Hackett 2010] provides a similar mechanism, except that it requires that all the variables in a formula are identified by terms with the same type; this severely limits its usability and is too restrictive for our case studies.

Our case studies exercise a range of the SMT-LIB standard and demonstrate the richness of our formula language. The case studies variously use algebraic data types and uninterpreted functions

[16] To get around this, our implementation uses a less precise rule that drops the negated premise.

(the Dminor type checker and the bottom-up points-to analysis); bit vectors and arrays (the Dminor type checker and the symbolic evaluator); and integers, uninterpreted sorts, and quantifiers (the Dminor type checker). It is easy to extend Formulog with additional theories (by adding new constructors) and different types of logical reasoning (by adding new operators, like `interpolate`). As Formulog so loosely couples Datalog evaluation and constraint solving, it is easy to swap in new solver backends without major changes to the Formulog runtime; our prototype currently supports Z3 [de Moura and Bjørner 2008], CVC4 [Barrett et al. 2011], and Yices [Dutertre 2014].

The design of Formulog makes it possible to advantageously apply Datalog-style optimizations to SMT-based analyses, with the result that Formulog programs can compete with analyses written in more mature languages. All of our case study implementations benefit from automatic parallelization: this scales our Dminor type checker and symbolic evaluator over the reference implementations, and helps our bottom-up points-to analysis be reasonably performant. The points-to analysis and symbolic evaluator also demonstrate the potential of the magic set transformation, as we have used it to derive demand-driven versions of these SMT-based analyses. While these types of optimizations could be added by hand to the reference implementations we compare against, the point is that the design of Formulog means that Formulog-based analyses get these optimizations for free, without the explicit effort of the analysis designer. Moreover, because of Formulog's close affinity to Datalog, a Formulog runtime can be augmented with additional Datalog-style optimizations. For instance, a Formulog runtime could use an incremental Datalog evaluation algorithm, which efficiently evaluates Datalog programs while facts are added or retracted from EDB relations [Gupta et al. 1993; Szabó et al. 2018]. This would be helpful for using SMT-based analyses in situations where the code under analysis changes, such as in IDEs or rapidly evolving codebases.

It speaks to the design of Formulog that the high-level optimizations it enables can, in many cases, make up for the naivety of our prototype runtime. Nonetheless, we are optimistic that significantly better performance can be achieved with a sophisticated backend. As we have designed Formulog to be close to Datalog, we can take advantage of many of the optimizations that have helped Datalog systems scale. For example, since we maintain the range restriction (which entails that every derived fact is ground), we can use concurrent data structures specialized for Datalog evaluation [Jordan et al. 2019]; since Formulog can be evaluated using standard semi-naive evaluation, we can compile Formulog programs to C++ following Soufflé's strategy [Jordan et al. 2016].

The ML fragment is an integral part of Formulog and has a substantial impact on its usability. As discussed in Section 3.2, the first-order fragment of ML we use can be translated in a pretty straightforward way to Datalog rules, and hence can be thought of as syntactic sugar. Despite this, the ML fragment is an integral part of the Formulog programming experience. First, it improves the ergonomics of Formulog, by making it more natural to manipulate complex terms. In particular, pattern matching and let expressions provide a structured way to reflect on complex terms and sequence computation on them; this same effect is not always as easy to achieve in Datalog rules. Second, it helps Formulog achieve its design goal of allowing SMT-based analyses to be implemented in a style close to their specification, since formal specifications often involve functions in addition to inference rules. Third, it improves the performance of Formulog, as there is more overhead involved with evaluating Datalog rules than evaluating an ML expression. A substantial amount of our case study code is in the ML fragment: The ratio of functions to rules is 1:4 for the bottom-up points-to analysis and 3:4 for the two other case studies (Table 3). Typically, the case studies use Horn clauses to define the overall structure of the analysis, and ML functions for structuring lower-level control flow, mirroring the use of judgments and helper functions in analysis specifications.

The limitation to first-order ML has several advantages. From a theoretical perspective, it means that there is an easy translation from it to Datalog rules, which allows us to give the standard

Herbrand model-based semantics to Formulog programs. From a practical perspective, it ensures that we never have to unify functions, which would require higher-order unification. The specifications of our case studies did not make heavy use of higher-order functions, so they were not much missed. However, a future version of Formulog could allow a limited use of higher-order functions (for example, those programs that can be compiled to the first-order fragment).

7 RELATED WORK

Datalog-based frameworks and domain-specific languages for static analysis. A variety of static analysis frameworks have been developed based on more-or-less standard Datalog, such as bddbddb [Whaley et al. 2005], Chord [Naik 2011], Doop [Bravenboer and Smaragdakis 2009], QL [Avgustinov et al. 2016], and Soufflé [Scholz et al. 2016]. Recent work has explored synthesizing Datalog-based analyses [Albarghouthi et al. 2017; Raghothaman et al. 2019]. Flix [Madsen et al. 2016] and IncA [Szabó et al. 2018] extend Datalog for analyses that operate over lattices besides the powerset lattice. IncA supports incremental evaluation, while Flix (like Formulog) includes algebraic data types and a pure functional language. Dataflow analysis is used as a case study for Datafun, a language combining Datalog and higher-order functional programming [Arntzenius and Krishnaswami 2016]. It might be possible to encode something like Formulog in Datafun; however, although it has recently been shown that Datafun can be evaluated using semi-naive evaluation [Arntzenius and Krishnaswami 2020], it is not clear to what extent other Datalog optimizations can be applied to Datafun programs. By combining Datalog with functional programming, Formulog, Flix, and Datafun are related to functional logic programming [Antoy and Hanus 2010]. The functional fragment of Formulog is less expressive than what is typically found in such languages, as Formulog functions are not first-class values and not higher-order.

Logic programming with constraints and formulas. The two dominant prior paradigms for combining logic programming and constraint solving are constraint logic programming (CLP) [Jaffar and Lassez 1987; Jaffar and Maher 1994] and constrained Horn clause (CHC) solving [Bjørner et al. 2015; Grebenshchikov et al. 2012; Gurfinkel et al. 2015; Hoder and Bjørner 2012]. As discussed in Section 6, these systems typically encode constraints as predicates, not terms, and thus support programming with constraints as things to be satisfied, rather than programming with formulas, which can be manipulated in more interesting ways (e.g., validity checking). In the context of static analysis, these systems have been used primarily for model checking, where a model of the input system is encoded using Horn clauses [Bjørner et al. 2015; Delzanno and Podelski 1999; Flanagan 2004; Fribourg and Richardson 1996; Grebenshchikov et al. 2012]. The rules depend on the program being analyzed, and the solutions to these rules reveal properties of the model; e.g., SeaHorn [Gurfinkel et al. 2015] checks programs by solving a CHC representation of their verification conditions. This differs than the approach taken in this paper, where the rules encode an analysis independent of the input program. The Datalog mode of μZ [Hoder et al. 2011] can be thought of as a bottom-up CLP system with special support for abstract interpretation.

A few existing logic programming systems support programming with formulas (vs constraints); we would argue that none do so with the same richness and flexibility as Formulog. Codish et al. [2008] extend Prolog with an interface to a SAT solver. SICStus Prolog [Carlsson and Mildner 2012], with its CLP extensions, has been used to write model checkers [Delzanno and Podelski 1999; Fribourg and Richardson 1996; Grebenshchikov et al. 2012; Podelski and Rybalchenko 2007]; these implementations typically rely on Prolog's non-logical features, like `assert`, making it harder to apply high-level optimizations like parallelization. Calypso [Aiken et al. 2007; Hackett 2010] is a Datalog variant that interfaces with external constraint solvers and has specialized support for bottom-up analyses. Calypso has been used with SAT and integer constraint solvers; in theory, it

could be connected to an SMT solver. However, Formulog offers several advantages over Calypso for SMT-based analyses. First, Formulog's approach to constructing formulas (via complex terms) and manipulating them (via its ML fragment) scales to the complex and heterogeneous formulas that arise in the SMT context, whereas Calypso's approach to formulas (opaque terms, constructed via predicates) would be cumbersome in this setting. Second, Formulog's type system supports the construction of expressive (and safe) formulas involving user-defined terms such as algebraic data types and uninterpreted functions. Third, the ML fragment of Formulog goes a long way towards making it practical for SMT-based analyses, by closing the gap between specification and implementation, and improving ergonomics and performance.

The logic programming language λProlog provides a natural way to represent logical formulas using a form of higher-order abstract syntax based on λ-terms and higher-order unification [Miller and Nadathur 1987; Pfenning and Elliott 1988]. Although this representation simplifies some aspects of using formulas, moving to a higher-order setting would complicate Formulog, widen the gap between Formulog and other Datalog variants, and potentially be an impediment to building a performant and scalable Formulog implementation. Answer set programming (ASP) uses specialized solvers to find a *stable model* (if it exists) of a set of Horn clauses [Brewka et al. 2011; Gelfond and Lifschitz 1988]. Common extensions support constraints on the shape of the stable model that will be found. ASP enables concise encoding of classic NP-complete constraint problems such as graph k-coloring, but it is not as obviously applicable to static analysis problems.

Type system engineering. PLT Redex [Felleisen et al. 2009] and Spoofax [Kats and Visser 2010] support exploratory type system engineering. PLT Redex supports a notion of judgment modeled explicitly on inference rules. Spoofax's type engineering framework, Statix, uses a logic programming syntax to specify type systems, with a custom solver for resolving the binding information in scope graphs simultaneously with solving typing constraints [van Antwerpen et al. 2018]. Both of these systems use custom approaches to finding typing derivations; neither supports SMT queries, but Statix's custom solver can resolve constraint systems that might not always terminate in Formulog.

Solver-aided languages. Scala^{Z3} [Köksal et al. 2011] supports mixed computations combining normal Scala evaluation and Z3 solving; we avoid this level of integration. Smten [Uhler and Dave 2013] is a solver-aided language that supports both concrete and symbolic evaluation; Rosette [Torlak and Bodik 2013] is a framework for creating solver-aided languages that have this property.

8 CONCLUSION

Formulog is a domain-specific language for writing SMT-based static analyses that judiciously combines Datalog, ML, and SMT solving (via an external SMT solver). As demonstrated by our case studies, it makes it possible to concisely implement a range of SMT-based analyses — refinement type checking, bottom-up points-to analysis, and symbolic evaluation — in a way close to their formal specifications, while also making it possible to automatically and advantageously apply high-level optimizations to these analyses like parallelization and goal-directed rewriting.

ACKNOWLEDGMENTS

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). We thank Arlen Cox, Scott Moore, the Harvard PL group, and anonymous reviewers for thoughtful feedback on earlier drafts.

REFERENCES

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

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming*. 689–706. https://doi.org/10.1007/978-3-319-66158-2_44

Sergio Antoy and Michael Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (2010), 74–85. https://doi.org/10.1145/1721654.1721675

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

Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1371–1382. https://doi.org/10.1145/2723372.2742796

Michael Arntzenius and Neel Krishnaswami. 2020. Seminäive 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

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: A Functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 214–227. https://doi.org/10.1145/2951913.2951948

Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-Oriented Queries on Relational Data. In *Proceedings of the 30th European Conference on Object-Oriented Programming*. 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

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

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 Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 1–15. https://doi.org/10.1145/6012.15399

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). 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

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis (Extended Version). arXiv:2009.08361 [cs.PL]

Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, 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

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II*. Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 243–262. https://doi.org/10.1145/1640089.1640108

Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. 2011. Answer Set Programming at a Glance. *Commun. ACM* 54, 12 (2011), 92–103. https://doi.org/10.1145/2043174.2043195

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.

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

Mats Carlsson and Per Mildner. 2012. SICStus Prolog–The First 25 years. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 35–66. https://doi.org/10.1017/S1471068411000482

Alessandro Cimatti and Alberto Griggio. 2012. Software Model Checking via IC3. In *Proceedings of the 24th International Conference on Computer Aided Verification*. 277–293. https://doi.org/10.1007/978-3-642-31424-7_23

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

Michael Codish, Vitaly Lagoon, and Peter J Stuckey. 2008. Logic Programming with Satisfiability. *Theory and Practice of Logic Programming* 8, 1 (2008), 121–128. https://doi.org/10.1017/S1471068407003146

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 238–252. https://doi.org/10.1145/512950.512973

William Craig. 1957. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic* 22, 3 (1957), 269–285. https://doi.org/10.2307/2963594

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

Giorgio Delzanno and Andreas Podelski. 1999. Model Checking in CLP. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 223–239. https://doi.org/10.1007/3-540-49059-0_16

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (2005), 365–473. https://doi.org/10.1145/1066100.1066102

Bruno Dutertre. 2014. Yices 2.2. In *Proceedings of the 26th International Conference on Computer Aided Verification*. 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex* (1st ed.). The MIT Press.

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

Cormac Flanagan. 2004. Automatic Software Model Checking via Constraint Logic. *Science of Computer Programming* 50, 1-3 (2004), 253–270. https://doi.org/10.1016/j.scico.2004.01.006

Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium*. 1075–1092.

Laurent Fribourg and Julian Richardson. 1996. Symbolic Verification with Gap-Order Constraints. In *Proceedings of the 6th International Workshop on Logic Programming Synthesis and Transformation*. 20–37. https://doi.org/10.1007/3-540-62718-9_2

Hervé Gallaire and Jack Minker (Eds.). 1978. *Logic and Data Bases*. Plenum Press.

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*. 1070–1080.

Sergey Grebenshchikov, Nuno Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 405–416. https://doi.org/10.1145/2254064.2254112

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 and Trends in Databases* 5, 2 (2013), 105–195. https://doi.org/10.1561/1900000017

Salvatore Guarnieri and V Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Proceedings of the 18th USENIX Security Symposium*. 78–85.

Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining Views Incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166. https://doi.org/10.1145/170035.170066

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn Verification Framework. In *Proceedings of the 27th International Conference on Computer Aided Verification*. 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

Brian Hackett. 2010. *Type Safety in the Linux Kernel*. Ph.D. Dissertation. Stanford University.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 58–70. https://doi.org/10.1145/503272.503279

Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*. Springer, 157–171. https://doi.org/10.1007/978-3-642-31612-8_13

Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. 2011. μZ–An Efficient Engine for Fixed Points with Constraints. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. 457–462. https://doi.org/10.1007/978-3-642-22110-1_36

Joxan Jaffar and Jean-Louis Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 111–119. https://doi.org/10.1145/41625.41635

Joxan Jaffar and Michael J. Maher. 1994. Constraint Logic Programming: A Survey. *The Journal of Logic Programming* 19 (1994), 503–581. https://doi.org/10.1016/0743-1066(94)90033-7

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Proceedings of the 28th International Conference on Computer Aided Verification*. 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. 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

Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the 25th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 444–463. https://doi.org/10.1145/1869459.1869497

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2011. Scala to the Power of Z3: Integrating SMT and Programming. In *Proceedings of the 23rd International Conference on Automated Deduction*. 400–406. https://doi.org/10.1007/978-3-642-22438-6_30

Robert Kowalski. 1979. Algorithm = Logic + Control. *Commun. ACM* 22, 7 (1979), 424–436. https://doi.org/10.1145/359131.359136

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization*. 75–88. https://doi.org/10.1109/CGO.2004.1281665

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The Size-Change Principle for Program Termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 81–92. https://doi.org/10.1145/360204.360210

V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*. 271–286.

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

Kenneth L McMillan. 2006. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*. Springer, 123–136. https://doi.org/10.1007/11817963_14

Dale Miller and Gopalan Nadathur. 1987. A Logic Programming Approach to Manipulating Formulas and Programs. In *Proceedings of the 1987 Symposium on Logic Programming*. 379–388.

Mayur Naik. 2011. Chord: A Program Analysis Platform for Java. https://www.seas.upenn.edu/~mhnaik/chord/user_guide/index.html. Accessed: 2020-04-01.

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. 199–208. https://doi.org/10.1145/53990.54010

Andreas Podelski and Andrey Rybalchenko. 2007. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*. 245–259. https://doi.org/10.1007/978-3-540-69611-7_16

Teodor C Przymusinski. 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

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2019. Provenance-Guided Synthesis of Datalog Programs. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–27. https://doi.org/10.1145/3371130

Thomas W. Reps. 1995. Demand Interprocedural Program Analysis Using Logic Databases. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*. 163–196.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 159–169. https://doi.org/10.1145/1375581.1375602

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

Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*. Springer, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

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

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 135–152. https://doi.org/10.1145/2509578.2509586

Petar Tsankov, Andrei Dan, Dana Drachsler-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

Richard Uhler and Nirav Dave. 2013. Smten: Automatic Translation of High-Level Symbolic Computations into SMT Queries. In *Proceedings of the 25th International Conference on Computer Aided Verification*. 678–683. https://doi.org/10.1007/978-3-642-39799-8_45

Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 114:1–114:30. https://doi.org/10.1145/3276484

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

Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

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. https://doi.org/10.1007/11575467_8

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