# Datalog + SMT for Static Analysis

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

## Abstract

Datalog has been successfully used as the implementation language for a range of static analyses. However, current variants of Datalog do not naturally support static analyses that use logical formulas, a rich class of analyses that includes refinement type checking, symbolic execution, and many forms of model checking. This paper proposes Formulog, a Datalog variant designed to fill this gap by making it easy to represent and reason about logical formulas. Formulog augments Datalog evaluation with the ability to call out to an external satisfiability modulo theories (SMT) solver. Despite this added expressiveness, Formulog is designed so that Formulog programs can still be optimized using powerful Datalog optimizations and evaluated using scalable Datalog algorithms. Our case studies demonstrate that a diverse range of formula-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.

## 1 Introduction

The logic programming language Datalog offers concrete benefits to static analysis implementors. Embodying Kowalski's principle of separating the logic of a computation from the control necessary to perform that computation [47], Datalog frees analysis designers from low-level implementation details and enables them to program at the specification level (such as at the level of formal inference rules). Consequently, Datalog-based analyses can be orders of magnitude more concise than counterparts written in more traditional languages [72]. What is more, the high-level nature of Datalog makes it amenable to high-level optimizations, such as automatic parallelization and global program rewriting. These optimizations can not only help make Datalog-based analyses acceptably performant — in fact, at times Datalog-based analyses have outperformed the non-Datalog state-of-the-art [18] — but can also be used to synthesize previously-unimplemented analyses "for free," such as demand-driven versions of exhaustive analyses [58].

A wide range of analyses have been implemented in Datalog and its variants, such as dataflow-style analyses [7, 52], points-to analyses for object-oriented languages [18, 73], and security analyses for Java [42, 51], JavaScript [33], and Ethereum smart contracts [31, 67]. However, an important class of analyses cannot be naturally encoded in current Datalog variants: analyses that use logical formulas. Such analyses include refinement type checking, abstract interpretation over the predicate domain, symbolic execution, and various forms of model checking. These analyses often symbolically represent reachable program states as logical formulas, and then reason about them in theories that support concepts such as machine integers and arrays. No Datalog variant currently provides language abstractions for computing over these types of formulas, and thus this class of analyses cannot take advantage of the benefits of Datalog.

```
The program has a DBZ at line L if...
div_by_zero(L) :-
  ...the interpreter reaches line L with state St
  reachable(L, St),
  ...L is the instruction Res := Num / Den and
  div_inst(L, Res, Num, Den),
  ...an SMT encoding of possible values of Den
  Enc_den = encode(Den),
  ...indicates Den can be zero.
  is_sat(`St /\ Enc_den #= 0`).
```

**Figure 1.** This Formulog rules defines when a symbolic evaluator has uncovered a possible division-by-zero error.

This paper presents Formulog, a Datalog variant that fills this gap by making it easy to represent logical formulas and reason about them as formulas. Like a Datalog program, a Formulog program consists of a set of rules that define relations between ground (i.e., variable-free) terms. Some of these terms are interpreted as logical formulas when they are passed as arguments to built-in functions. These functions can be invoked from the bodies of Formulog rules; the functions are implemented by calls to an external satisfiability modulo theories (SMT) solver.

For a concrete example, consider the rule in Figure 1, which describes one case in symbolic evaluation, an analysis where an interpreter evaluates a program while maintaining a state that symbolically represents the current environment. Here we assume the symbolic state is a proposition (e.g., a conjunction of constraints over program variables). This rule identifies a division-by-zero bug at program location `L` if the evaluator has reached that location with state `St`, if that

*Work done while on sabbatical at Harvard University.

*PL'18, January 01–03, 2018, New York, NY, USA*
2018.

location contains a division instruction with divisor `Den`, and if the symbolic encoding of `Den` can be zero given `St`. The logical term `` `St /\ Enc_den #= 0` `` represents the conjunction of the current state and the constraint that (the symbolic encoding of) the divisor is zero; the SMT interface function `is_sat` tests the logical satisfiability of this proposition. The analysis would also need to separately define the function `encode` and the predicates `reachable` and `div_inst`.

Formulog is designed to be compatible with more traditional variants of Datalog, both in terms of semantics and evaluation techniques: A Formulog program can be understood as a minimal model of its rules, and can be evaluated using traditional Datalog algorithms (with hooks for SMT calls). This means not only that Formulog retains many of the benefits of Datalog, but also that it is easier to port additional language features and optimization techniques from the Datalog literature to Formulog. However, keeping Formulog "close" to Datalog entails breaking with previous paradigms integrating logic programming and constraint solving, such as constraint logic programming [40] and constrained Horn clause solving [15]. In particular, whereas previous paradigms have closely wed the evaluation of logic programming rules with the solving of constraints, Formulog intentionally keeps the two distinct.

In essence, Formulog can be seen as a Datalog variant that provides a nice interface to an SMT solver. To enable this interface, Formulog extends Datalog with algebraic data types and first-order functions, and uses a bimodal type system that treats terms appearing in logical formulas more liberally than terms appearing outside of formulas, while still ensuring that only well-typed formulas are constructed. The bimodal nature of the type system reflects the two sides of Formulog evaluation, as during normal evaluation it distinguishes between concrete values and symbolic values (e.g., between `42` and a formula representing the sum of two integers), but conflates concrete and symbolic values during constraint solving, when the distinction is not meaningful.

The overall design of Formulog is based on the hypothesis that the simple Horn clause logic of Datalog-style rules is sufficient for describing the high-level control flow of many analyses that use logical formulas, and that more complex (and computationally expensive) constraint solving mechanisms only need to be invoked locally on demand. To test this hypothesis, we have implemented three diverse formula-based analyses using Formulog: a type checker for a sophisticated refinement type system, a bottom-up context-sensitive points-to analysis for Java, and a bounded symbolic evaluator for a simple imperative language. Our Formulog implementations of these analyses are concise (up to 10x smaller than the reference implementations), and – despite the naivety of our prototype Formulog runtime – they are all fast enough to be useful. In fact, for every case study, the Formulog version was in some instances *faster* than the reference implementation.

| | | | |
|---|---|---|---|
| Programs | prog | ::= | $H^*$ |
| Horn clauses | $H$ | ::= | $A \mathrel{:-} P_1, \ldots, P_n.$ |
| Premises | $P$ | ::= | $A \mid\, !A$ |
| Atoms | $A$ | ::= | $p(t_1, \ldots, t_n)$ |
| Terms | $t$ | ::= | $X \mid c(t_1, \ldots, t_n)$ |
| Variables | $X$ | $\in$ | Var |
| Constructor symbols | $c$ | $\in$ | CSym |
| Predicate symbols | $p$ | $\in$ | PSym |

**Figure 2.** A Datalog program consists of a set of Horn clauses that define relations between ground (variable-free) terms.

Our case studies demonstrate that Formulog can deliver powerful high-level optimizations to complex analyses. All of our implementations benefit from automatic parallelization. Furthermore, our symbolic evaluator and bottom-up points-to analysis are *goal-directed*, and focus on exploring only the parts of an input program relevant to answering a specific query. We were able to derive these goal-directed analyses for free via the magic set transformation, an automated program rewriting technique [11] that we applied to the exhaustive versions of the analyses (which are presumably easier to write than the goal-directed ones).

In sum, this paper makes the following contributions:

- It proposes a mechanism to integrate Datalog evaluation with SMT solving that retains many of the traditional benefits of programming with Datalog.
- It presents a lightweight bimodal type system that mediates the interface between Datalog evaluation and SMT solving. By treating terms inside logical formulas more liberally than terms outside of formulas, this type system makes it possible to construct expressive formulas while preventing many kinds of runtime errors in both Datalog evaluation and SMT solving.
- It shows, through case studies, that a range of formula-based analyses can be naturally and concisely encoded in Formulog, and that high-level Datalog optimizations can be advantageously applied to such programs.

## 2 Background and motivation

This section gives background on Datalog, describes why it is useful for writing static analyses, and motivates why we are excited to extend it with support for logical formulas.

### 2.1 Datalog

The starting point for Formulog is Datalog with constructors and stratified negation (Figure 2) [28, 32]. A Datalog program is defined by a set of Horn clauses, where a *clause* consists of a head atom and a set of body premises. A *premise* is a positive atom $A$ or a negated atom $!A$. An *atom* $A$ is a predicate symbol applied to a list of terms, where a *term* $t$ is a variable $X$ or a constructor symbol $c$ applied to a list of terms. Each predicate symbol 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).

Datalog has an elegant logical interpretation: A clause

$$A :- P_1, \ldots, P_m.$$

corresponds to the first-order logic formula

$$\forall X_1, \ldots, X_n.\, (P_1 \land \cdots \land P_m \implies A)$$

where $X_1, \ldots, X_n$ are the variables that occur in the clause. The semantics of a program is a distinguished, minimal Herbrand model of its clauses, which is guaranteed to exist thanks to syntactic restrictions like stratification (no cyclic negative dependencies between relations) [4, 32, 57, 70]. This means that a Datalog program can be evaluated via a fixpoint computation that finds the consequences of its rules.

Although Datalog traditionally limits ground terms to constants, we allow complex terms (i.e., non-nullary constructors). This follows the lead of many recent Datalog variants, including Soufflé [60], LogicBlox [5], and Flix [52]. In our context, complex terms provide a natural way to reify logical formulas, and they also can be used to create data structures such as lists that make it more natural to encode certain analyses. This comes with the cost of possibly-diverging programs: Adding *n*-ary constructors to Datalog makes it Turing-complete instead of polynomial time [32].

## 2.2 Datalog for static analysis

Datalog is a natural and effective way to encode many static analyses. EDB relations are used to represent the program under analysis (e.g., a relation could encode a control flow graph 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. The Datalog program will compute the contents of the IDB relations, which can be thought of as the analysis results.

When using Datalog, an analysis designer is effectively working at the level of logical specification; the Datalog runtime executes this specification while applying optimizations such as parallelization and goal-driven program rewriting. Not only can this make it easier to write analyses, but it can also make it easier to write *better* analyses because the analysis designer can focus on the high-level logic of the analysis; e.g., some designers have reported that writing their analysis in Datalog allowed them to more easily remedy algorithmic flaws that were making their analysis inefficient [62].

Additionally, Datalog can be a good platform for combining analyses, since logical predicates provide a flexible interface for tying code together. For example, Bravenboer and Smaragdakis [17] have used this aspect of Datalog to weave together analyses in a way that makes the composite analysis more effective than the analyses run separately.

```
The MOP analysis reaches location Loc with state M_state if...
mop_reach(Loc, M_state) :-
  ...the symbolic executor has reached Loc with state S_state
  symex_reach(Loc, S_state),
  ...Loc is an application program point and
  in_application(Loc),
  ...M_state is the translation of S_state.
  translated_state(S_state, M_state).

The symbolic executor reaches location Loc with state S_state if...
symex_reach(Loc, S_state) :-
  ...the MOP analysis has reached Loc with state M_state
  mop_reach(Loc, M_state),
  ...Loc is a framework program point and
  in_framework(Loc),
  ...S_state is the translation of M_state.
  translated_state(S_state, M_state).
```

**Figure 3.** These rules tie together a meet-over-all-paths (MOP) analysis and a symbolic executor in a hypothetical hybrid analysis of a framework-based Java web application.

## 2.3 Motivating Formulog

While expressive enough to support some analyses, Datalog is a very restricted language and many analyses cannot or cannot easily be encoded in it. Recent work has explored variants of Datalog that make it easier to write analyses that operate over interesting lattices [52, 64]. Following in this spirit, we explore how to extend Datalog with support for analyses that operate over logical formulas. Doing so has two main benefits. First, analyses that use logical formulas can now enjoy the advantages of Datalog. Second, by expanding the pool of analyses that can be composed via Datalog, we have potentially enabled novel composite analyses.

For example, Toman and Grossman [65] suggest a hybrid approach for analyzing framework-based Java web applications: A scalable meet-over-all-paths (MOP) analysis is used for the application code, and a variant of symbolic execution – being more precise but less scalable – is used for the framework code (which often uses language features like reflection that require high precision). As control flow goes back and forth between the application and the framework, these analyses interact in a mutually recursive way. This is the perfect setting for a Datalog-like language that makes it easy to encode interdependent analyses. In fact, given a symbolic executor and a MOP analysis, one could imagine tying them together with just a few rules (Figure 3). In a perfect world, composing these analyses would require no changes to either the MOP analysis or the symbolic executor. There could still be challenges to making an effective hybrid analysis. However, by using a Datalog-like language, we have removed the substantial implementation-level challenge of coordinating communication between the analyses.

This analysis is just one example of the type of composite analyses that would be possible if we had a Datalog variant that supports formula-based analyses (such as symbolic execution). This is precisely what Formulog sets out to do!

## 3 Language design

The design of Formulog is driven by two main desiderata. First, 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, and generate models of them. Second, we want to retain many of the traditional benefits of Datalog: We should still be able to apply Datalog optimizations to Formulog programs and evaluate them using Datalog algorithms.

In this section, we describe our design and how it supports the first desideratum; in Section 4, we demonstrate that it meets the second one via our prototype implementation and case studies; in Section 5, we go into more depth on how our approach differs from previous paradigms combining logic programming and constraint solving.

Here, we begin with an overview of the (non-formula) language features Formulog adds to Datalog, discuss how these features are used to support logical formulas, and then conclude by sketching Formulog's type system.

### 3.1 Basics

Absent logical formulas, a Formulog program essentially looks like ML-flavored Datalog code (Figure 4). In addition to built-in types such as strings, signed machine integers, floating point numbers, and tuples, users can define algebraic data types and records. They can also define ML-style functions, which are limited to being first-order and are not first-class values. These functions can be invoked in other functions and in logic rules. We support polymorphism and (mutual) recursion in both types and functions.

Functions in Formulog are call-by-value, which means that their arguments need to be normalized and ground (i.e, variable-free) by the time they are invoked. The Formulog runtime will rewrite rule bodies so that during evaluation any variable used in a function call is already bound to a value before the call site is evaluated; it will reject any rule that cannot be rewritten this way. For example, the runtime might have to reorder the premises in a rule body. This rewriting is safe so long as function calls do not diverge (an assumption that Formulog makes).

Adding first-order functions to Datalog is not foundational, as there is a relatively straightforward translation from Formulog's functions to Datalog rules. Thus, functions could be treated as just syntactic sugar, and a Formulog program could be understood in terms of the standard minimal model semantics. However, we believe the addition of functions greatly improves the ergonomics of Formulog: In our opinion, manipulating complex terms is often more natural in ML than in Datalog.[1] 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, since there is no order within a rule or between rules.

```
type 'a tree = lf | nd('a tree,'a,'a tree)

fun size(Tree: 'a tree) : bv[32] =
  match Tree with
  | lf => 0
  | nd(L,_,R) => 1 + size(L) + size(R)
  end

input num_tree(bv[32] tree)
num_tree(nd(lf,42,lf)).
num_tree(nd(nd(lf,10,lf),30,nd(lf,50,lf))).

output num_tree_size(bv[32] tree, bv[32])
num_tree_size(Tree, Sz) :-
  num_tree(Tree),
  size(Tree) = Sz.
```

**Figure 4.** A Formulog program consists of type definitions, function definitions, relation declarations, and Horn clauses. This program defines a polymorphic tree type and function that computes the size of a tree, tabulates an input relation of bit vector-valued trees, and then defines an output relation relating a tree (from the input relation) to its size.

Every predicate symbol in Formulog must be declared as either an `input` (EDB) relation or `output` (IDB) relation, and must be annotated with the types of its arguments, which cannot be polymorphic.[2] As in Datalog, `input` relations are enumerated explicitly as a set of facts, while `output` relations are defined using logic rules. The ML fragment of Formulog code can query into relations by "invoking" the relation as a function using the special wildcard term ?? as an argument. Given the relations from the example in Figure 4, the term `num_tree_size(??, ??)` evaluates to a list of pairs constituting that relation, and `num_tree_size(??, 42)` evaluates to a list of those trees in the relation with size 42. This mechanism provides a flexible way for users to define aggregation operations over relations. To maintain stratification, we require that any predicate that transitively makes this type of query is in a higher stratum than the queried relation.

[1]Speaking anecdotally, our case studies use logic rules to define the overall structure of the analysis, and ML functions for lower-level operations.
[2]We omit relation type declarations in many of the examples in this paper.

| | | | |
|---|---|---|---|
| Negation | `~☐` | : | `bool smt` ⟶ `bool smt` |
| Conjunction | `☐ /\ ☐` | : | `bool smt` ∗ `bool smt` ⟶ `bool smt` |
| Implication | `☐ ==> ☐` | : | `bool smt` ∗ `bool smt` ⟶ `bool smt` |
| Equality | `☐ #= ☐` | : | `'a smt` ∗ `'a smt` ⟶ `bool smt` |
| Variable creation | `#{ ☐ }[`$\tau$`]` | : | `'a` ⟶ $\tau$ `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` |
| Floating point addition | `fp_add` | : | `fp[`$j,k$`] smt` ∗ `fp[`$j,k$`] smt` ⟶ `fp[`$j,k$`] smt` |

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

| | | | |
|---|---|---|---|
| Satisfiability | `is_sat` | : | `bool smt` ⟶ `bool` |
| | `is_sat_opt` | : | `bool smt` ∗ `bv[32] option` ⟶ `bool option` |
| Validity | `is_valid` | : | `bool smt` ⟶ `bool` |
| | `is_valid_opt` | : | `bool smt` ∗ `bv[32] option` ⟶ `bool option` |
| Model generation | `get_model` | : | `bool smt` ∗ `bv[32] option` ⟶ `model option` |
| Model inspection | `query_model` | : | `'a sym` ∗ `model` ⟶ `'a option` |

**Figure 6.** Formulog provides built-in functions for reasoning about logical terms.

## 3.2 Logical formulas

Formulog uses data types and functions 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 functions 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 [10], as well as the theory of strings shared by the SMT solvers Z3 [23] and CVC4 [9].

### 3.2.1 Representing formulas

Users create logical terms through built-in 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 an infix constructor representing implication. As in this example, formulas may be quoted with backticks; we explain Formulog's quotation semantics in Section 3.3.

Formulog offers around 70 constructors for creating logical terms ranging from symbolic string concatenation to logical quantifiers annotated with patterns for trigger-based instantiation [25]. Figure 5 shows a sample of these constructors and their types. The formula type $\tau$ `smt` represents a $\tau$-valued formula term, and the type $\tau$ `sym` represents a $\tau$-valued formula variable, where (in each case) $\tau$ is a non-formula type. Thus, the type of a proposition is `bool smt`. The types for bit vectors and floating point are indexed by, respectively, the bit vector width and the widths of the exponent and significand. Some constructors also require explicit indices, such as `bv_const[`$k$`]`, which creates a symbolic $k$-bit wide bit vector value from a concrete 32-bit bit vector.

Formulog distinguishes between logic programming variables and formula variables. A formula variable is a ground term that, when interpreted logically, represents a symbolic value. The constructor `#{`$t$`}[`$\tau$`]` is used to create a formula variable of type $\tau$ `sym` identified by the term $t$ (which can be of arbitrary type). Intuitively, $t$ is the "name" of the variable. Because of this, the variable represented by `#{`$t$`}[`$\tau$`]` is guaranteed not to occur in $t$; i.e., it is fresh with respect to the set of formula variables in $t$. For example, if `X` is bound to a list of terms of type `bool sym`, the term `#{X}[bool]` is a new symbolic boolean variable that will not unify with any variable in `X`. The shorthand `#`*id*`[`$\tau$`]` is equivalent to `#{"`*id*`"}[`$\tau$`]`, where *id* is a syntactically valid identifier.

Finally, Formulog provides straightforward mechanisms for declaring uninterpreted sorts (essentially a special kind of type) and uninterpreted functions (essentially a special kind of constructor) that can be used within logical formulas.

### 3.2.2 Using formulas

One can reason about logical terms as formulas using the built-in functions in Figure 6. When a function in the SMT interface is invoked, its formula argument is translated into the SMT-LIB format and a call is made to an external SMT solver. These functions 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 function calls.

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 functions, the SMT interface functions can be invoked from the bodies of rules, as here:

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

This rule derives the fact `ok`: The term `#x[bool]` is not unifiable with the term `#y[bool]`, but these terms both may and may not be equal when interpreted as logical variables via the function `is_sat`. (Boolean-valued functions like `is_sat` can syntactically appear as atoms in rule bodies.)

Formulog provides two sets of functions 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 functions `is_sat` and `is_valid` return booleans. In the case that the backend SMT solver is not be able to determine whether a formula $\phi$ is satisfiable, the term `is_sat`($\phi$) is treated as not unifying with any term.[3] On the other hand, `is_sat_opt`($\phi$) will return `none` if the solver times out or gives up. (The pair `is_valid` and `is_valid_opt` works the same way.) While we suspect that the simpler versions will be sufficient for most applications, the optional versions do allow applications to explicitly handle the "unknown" case if need be (e.g., pruning paths in symbolic execution). They also provide an argument position for an optional timeout.

The function `get_model` takes a proposition and an optional timeout; it returns a model for the proposition 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 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 model, $x$ will be assigned the value of $e$.

#### 3.2.3 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:

[3]Our prototype also supports a "hard exception" mode, which will kill execution in this case.

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

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 $\tau$, Formulog provides a constructor `#is_`$c$ of type $\tau$ `smt` $\longrightarrow$ `bool smt`. The proposition `#is_`$c(t)$ holds if the outermost constructor of $t$ is $c$. The second kind is an argument getter. If $c$ is a constructor for type $\tau$ with $n$ arguments of types $\tau_i$ for $1 \leq 1 \leq n$, Formulog will generate $n$ argument getters of the form `#`$c\_i$, where $c\_i$ has type $\tau$ `smt` $\longrightarrow \tau_i$ `smt`. Within a formula, the term `#`$c\_i(t)$ represents the value of the $i^{\text{th}}$ argument of $t$. For example, we can construct a formula asserting 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 functions from Figure 6 to find a model of this (satisfiable) formula; in this model, `#x[bool list]` might be assigned the concrete value `cons(true, nil)`.

### 3.3 Type system

Formulog's type system is designed to meet three desiderata. The first desideratum is that Formulog evaluation should never go wrong, which could happen if, e.g., a term of type `string` were passed to a function expecting a `bool`. The second desideratum is that it should be possible to construct only logical terms that represent well-sorted formulas under the SMT-LIB standard; for example, it should be impossible to construct a term that represents the addition of a 32-bit bit vector with a 16-bit 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 (i.e., a bit vector-valued formula) since a function 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.**

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

This program would get stuck evaluating `f(bar(X))`, since `Y` is bound to a symbolic value in `f` but the 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.**
```
ok :-
  X = #x[bv[32]],
  is_valid(`bar(X) #= bar(5)`).
```

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 quotations. Within this system there are three sorts of types.[4] The first sort is non-logical types $\tau$, such as primitive types like `bv[32]` and user-defined algebraic data types. The second sort $\tau$ `smt` represents $\tau$-valued formulas; for example, `bv[32] smt` is the type of 32-bit bit vector-valued formulas. The third sort $\tau$ `sym` represents $\tau$-valued logical variables; for instance, the term `#x[bv[32]]` has type `bv[32] sym`. (We distinguish between $\tau$ `smt` and $\tau$ `sym` because some logical terms explicitly require an argument to be a logical variable, such as a quantifier binding a variable.) In essence, the Formulog type checker differentiates between these three sorts of types outside of quotations, but conflates them within quotes. This bimodal approach disallows Example 1 while permitting Example 2.

The type checker uses this behavior when checking terms quoted by backticks, and every quoted term is treated as having an `smt` type (so `true` has type `bool`, but `` `true` `` has type `bool smt`). There are two subtleties to type checking quoted terms. First, function calls that take arguments are disallowed from quotations; so, function arguments always have to check at the expected type. (Alternatively, we could allow such function calls "inside" quotations, but implicitly or explicitly unquote them.) Second, the type checker never conflates $\tau$ `sym` with other types in constructors that bind formula variables; e.g., the `forall` constructor needs a formula variable for its first argument, not an arbitrary formula.

Intuitively, this bimodal approach is safe because it distinguishes between concrete and symbolic values during normal evaluation (where conflating them might lead to going wrong), and conflates them only during SMT evaluation (where the distinction between concrete and symbolic values is not meaningful). In supplemental material accompanying this submission, we have formalized the Formulog type system and proven it sound with respect to the operational semantics of Formulog. The soundness of the type system with respect to the semantics of SMT-LIB follows from the fact that the types of the formulas provided by Formulog are consistent with the types given by the SMT-LIB standard, and that the Formulog type system guarantees that, at runtime, terms and their constituent formulas are well-typed.

[4]Here we gloss over a fourth sort, `model` (the type of models returned by `get_model`), as models are not allowed to appear within formulas.

## 4 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.

### 4.1 Prototype

We have implemented a prototype of Formulog in around 14K lines of Java. The implementation works in five stages: parsing, type checking, rewriting, validation, and evaluation. If the user has supplied a particular query to solve, the program is specialized in the rewriting phase for this query. Our rewriting algorithm is based on a variant of the magic set transformation that preserves stratified negation (i.e., if the original program meets the requirements of stratified negation, then so does the rewritten one) [53]. Properties like stratified negation and range restriction are checked during the validation phase. To achieve parallelism, our evaluator uses a work-stealing thread pool; worker threads dispatch logical queries to a pool of Z3 instances [23].

Our focus has been on the design and expressiveness of Formulog, and not on the performance of our prototype. As we have designed Formulog to be close to Datalog, many of the optimizations that have helped Datalog engines scale can be applied to Formulog, such as optimal index selection [63], specialized concurrent data structures [43], and synthesis of C++ implementations of analyses specified in Datalog/Formulog [42]. There can be a huge performance gap between naive and sophisticated Datalog implementations, a concrete example being that Souffle [60] is 7.7x faster than our prototype in computing a four-million-edge graph transitive closure relation. Even so, our prototype has performed acceptably on our case studies. In each case study, we have timed our Formulog implementation of the analysis on a small number of inputs and compared against a reference implementation.[5] While we hesitate to draw conclusions from small samples, we are encouraged by the fact that for many cases our implementations are competitive with the reference implementations (and sometimes even faster!).

[5]For each case study, we use a tool to take a program in the input language (e.g., Java) and turn it into Formulog facts. We do not include these times, as they are typically quite short. Extracting facts from libraries can take a few minutes, but this needs to be done only once per library.

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

**Figure 7.** 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.

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 (v4.8.4). For each result, we report the median of three trials.

## 4.2 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 [14]. 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$^\sharp$ and 400 lines of SMT-LIB; we estimate that the functionality we implemented accounts for over two thousand of these lines.[6]

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 expressivity of Formulog's formula language. For example, Figure 7 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 as set of mutually-recursive functions that encode expressions, environment, and types. For example,

[6]The reference Dminor implementation is not open source, but the authors have kindly provided us with line counts for each source file.

```
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)
  | ...
```

**Figure 8.** This function constructs a formula capturing the logical denotation of a Dminor type.

```
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, Ax1),
  encode_type(T1, X) = (Phi_t1, Ax2),
  Axs = `axiomization /\ Ax1 /\ Ax2`,
  is_valid_opt(
    `Axs /\ Phi_env /\ Phi_t ==> Phi_t1`,
    z3_timeout) = some(true).
```

**Figure 9.** This rule defines Dminor's semantic subtyping relation. It uses the function `is_valid_opt` instead of `is_valid` because its SMT queries can sometimes result in "unknown."

the type encoding function (fragment, Figure 8) takes a type $\tau$ and an (encoded) Dminor value $v$, this function returns two propositions. The first is true when $v$ has type $\tau$. 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 9 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 [49]. Our implementation is another good example of the synergy 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 fixpoint 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). Our times ranged from 1.7-3.1 seconds; these were 2.0-4.1x slower than the reference implementation.[7] As an optimization, the reference implementation tries syntactic subtyping before semantic subtyping; when we disabled this feature (which we did not implement), our implementation was still up to 3.3x slower on small examples, but was slightly faster than the reference implementation on the most complex example. Additionally, when we ran the type checkers on a composite program consisting of all six examples concatenated together, the Formulog version had a speedup of 1.3x over the reference implementation (with syntactic subtyping turned off), suggesting that Formulog's automatic parallelization can help with programs that have many pieces that can be type checked independently.

***Other refinement type checkers*** We have implemented other refinement type systems: System FH, a core calculus with refinement types but no actual use of an SMT solver [12, 61]; and a bidirectional version of liquid type checking [59, 71]. The liquid type checker is about 350 LOC and is able to check programs written over refined booleans; adding other base types and logical qualifiers would not be challenging.

## 4.3 Bottom-up points-to analysis

We implemented the bottom-up context-sensitive points-to analysis for Java proposed by Feng et al. [27]. A points-to analysis computes an over-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. This bottom-up approach contrasts with top-down points-to analysis, which explores a program in the opposite direction,

[7]We ran the reference implementation on a machine with Microsoft Windows Server 2019 and the same hardware specs as our Ubuntu machine.

```
instant_ptsto(C, O1, Phi1, O2, Phi_all) :-
  instant_loc(C, hp(O1), hp(O2), Phi2),
  instant_constraint(C, Phi1, Phi3),
  Phi_all = conjoin(Phi2, Phi3).
```

**Figure 10.** This (slightly simplified) 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 `hp(O1)` can be instantiated to a heap location `hp(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 `Phi_all`, the conjunction of `Phi2` and `Phi3`.

starting at the entry points and moving from caller to callee. The main advantage with the bottom-up approach is that a summary only has to be computed once, whereas a top-down approach might analyze the same method multiple times in different calling contexts. While context-sensitive top-down points-to analyses have been one of the most successful applications of Datalog to static analysis, implementations of bottom-up algorithms have been much rarer (with the work of Hackett and Aiken [36] being one example).

In the algorithm proposed by Feng et al., a method summary consists of 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 annotated with logical formulas that describe the conditions under which the edges hold; the formulas are constraints on the runtime types of various heap objects (such as the receiver of a method invocation). A method summary is used at a call site by instantiating the abstract heap associated with that method. Their implementation, Scuba, is 16K lines of Java and is built on top of the Chord framework [50]. Our implementation is around 1,500 lines of Formulog, of which approximately 200 lines define a top-down context-insensitive points-to analysis that is used as a "pre-analysis" for the bottom-up one (Scuba uses the one in Chord).

Our implementation is concise because it closely follows the inference rules describing the algorithm given by Feng et al., with some adjustments to make the analysis more natural for our Formulog setting. For example, Figure 10 shows a rule describing how a points-to edge (i.e., a target object labeled with a constraint) is instantiated at a call site. This is one step in the process of instantiating a summary at a call site, a process involving complex logic defined through half a dozen mutually recursive relations. The Java code for encoding this logic is substantially more complex, verbose, and (we would argue) error-prone than the Formulog code.

We ran our implementation against the reference implementation on three mid-size input programs – a red-black tree implementation, a Prolog interpreter, and an HTTP

| Example | # summaries | Scuba | Formulog |
|---|---|---|---|
| Red-black tree | 60 | 1:42 | 4:26 |
| Prolog interpreter | 118 | 1:51 | 4:38 |
| HTTP server | 122 | 6:10 | 4:58 |
| Antlr | 834 | 2:51 | 28:06 |

**Table 1.** Our Formulog implementation of a context-sensitive bottom-up points-to analysis for Java was competitive with Scuba, the reference implementation, on mid-size programs, but lagged behind on the largest benchmark (times are in min:sec format). Both implementations summarized library methods using *context-insensitive* points-to information (we list the number of context-sensitive summaries).

server – as well as the DaCapo benchmark Antlr [16] (Table 1). On the smaller examples, we were up to 2.6x slower than Scuba, but also up to 1.2x faster. We completed on Antlr in less than 30 minutes; however, here we were 9.9x slower than Scuba. Although full parity might be hard to achieve, it should be possible to narrow this gap. First, Scuba achieves scalability through heuristics that are not fully documented in the paper; we have implemented only some of these. Second, having a sophisticated Formulog runtime could make a big difference for a workload of this scale.

The concision of the Formulog version – over 10x smaller than Scuba itself, not to mention Chord's codebase – and its proximity to the algorithm's formal specification suggest that it could be useful for prototyping something like Scuba. First, it could help uncover logical inconsistencies in the specification. For instance, because there is such a close correspondence between judgments in the specification and relations in the Formulog version, the Formulog type checker revealed that the type signature given for one judgment in Feng et al.'s formalization does not match its definition (a not-just-cosmetic inconsistency, as later rules depend on it having the stated type). Second, it could be a fruitful place to experiment with specification-level heuristics without the clutter of low-level implementation details. Third, because it is much closer to the specification than the Java version, it could be used as a standard to test against.

Moreover, the Formulog version can be run in a goal-directed mode: If a client needs the summaries of only a subset of methods, the analysis will compute only the summaries necessary for constructing the target summaries. The demand-driven version is derived automatically from the exhaustive analysis via the magic set transformation. Thus, even in cases where the Formulog version does not scale to analyzing the entire program, this feature could be used to compute summaries for a sample of methods and check a more scalable implementation against them.

```
failed_assert(Path, St) :-
  assert_instruction(Instr, X),
  stepped(Instr, St, _, Path),
  has_value(X, St, v_int(V)),
  !assert_not_zero(V, St).
```

**Figure 11.** This rule defines when the symbolic evaluator has reached a failing assert instruction: when the argument of the assert instruction may be zero.

```
a := new symbolic array of size N;
b := new symbolic value;
if (b) { sort a; }
else { sort a; verify a is sorted; }
```

**Figure 12.** This pseudocode sketches a C program that creates an array, branches, sorts it in each branch, but only verifies that the result is sorted in one branch.

## 4.4 Bounded symbolic evaluation

We have used Formulog to implement a symbolic evaluator for a fragment of LLVM bitcode [48] that corresponds to a While language with arrays. Our evaluator is around 800 LOC and is a mix of bounded model checking [13] and symbolic execution [45]: It explores all feasible program paths up to a given length, evaluating the program concretely whenever possible and aggressively pruning infeasible paths.

A symbolic evaluator avoids explicitly enumerating all possible inputs by treating inputs symbolically. A symbolic value represents a set of runtime values. The set associated with a symbolic value is initially unconstrained (it can concretely have any value of the relevant type). When the symbolic evaluator reaches a condition that depends on a symbolic value, 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. In each fork, it constrains the symbolic value so that it is consistent with the branch that has been taken. These constraints take the form of logical formulas over the symbolic values; the aggregate constraint over all symbolic values for a particular path taken by the evaluator is known as the *path condition.*

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 11). This rule says that the path `Path` ends in a failure with evaluator state `St` if there is an assert instruction `Instr` with argument `X`, following `Path` has led the evaluator to that instruction with state `St`, in that state `X` has the (possibly symbolic) integer value `V`, and `V` may be zero. The function `assert_not_zero(V, St)` returns true if and only if `V` cannot be zero given `St`.

| $N$ | # paths | KLEE | Flg-exh | Flg-dir |
|---|---|---|---|---|
| 5 | 388 | 0:15 | 0:14 (1.1x) | 0:11 (1.4x) |
| 6 | 2,718 | 2:13 | 1:49 (1.2x) | 1:24 (1.6x) |
| 7 | 22,070 | 23:31 | 16:28 (1.4x) | 12:19 (1.9x) |

**Table 2.** For various array lengths $N$ in the program in Figure 12, our symbolic evaluator was faster than KLEE. We give absolute times (min:sec) and speedup over KLEE for the exhaustive (Flg-exh) and the goal-directed (Flg-dir) versions. We also give the number of program paths for each array length; the directed version explores only a subset of these.

We have evaluated our symbolic evaluator on the C program sketched in Figure 12. This program creates an array of size $N$ filled with symbolic integers and then splits execution into two branches. The first branch just sorts the array using selection sort; the second sorts it and then verifies (using assertions) that every element in the array is no greater than the next element. Our symbolic evaluator can explore this program exhaustively, or we can instruct it to do a more directed exploration. If we only want to check that no assertion fails, we can add the query

```
:- failed_assert(_Path, _St).
```

The Formulog runtime will use the magic set transformation to rewrite our evaluator to explore only paths that could potentially lead to a failed assertion. After this transformation, our evaluator does not explore the first branch of the input program, substantially reducing the search space.

On this example, our evaluator is faster than the symbolic executor KLEE (v2.0) [20]. Our implementation is naive compared to KLEE; however, automatic parallelization and goal-directed rewriting lead to speed ups of up to 1.9x (Table 2). The comparison is not apples-to-apples (e.g., KLEE needs to maintain a more complex state), but the results nonetheless demonstrate the power of Datalog optimizations.

## 5 Comparison: CLP and CHC solving

Previous work has explored the integration of logic programming with constraint solving. This section draws out the differences between Formulog and two prominent examples: constraint logic programming (CLP) and constrained Horn clause (CHC) solving. The key difference is that, while these paradigms provide a way to program with *constraints*, Formulog provides a way to program with *logical formulas*.

In CLP [40, 41], rule bodies can contain constraints over logic programming terms; these constrain the search space during evaluation. Many CLP implementations are extensions of Prolog [29], and thus can take advantage of the fixed evaluation order of Prolog to strategically explore constraint satisfaction problems. Consider these clauses:

```
p(Y) :- Y < 2, q(Y), r(Y).
q(X) :- X > 0.
```

Assuming the constraints are over integers, a Prolog-style evaluator would know that `Y` must be bound to 1 by the time it reaches the premise `r(Y)` in the first rule, and thus could explore a smaller space when searching for answers to this subgoal. However, having a fixed evaluation order makes it hard to apply optimizations that have proved useful for scaling static analyses in Datalog, such as parallelization and database-style query planning [18, 60]. $\mu$Z's Datalog mode [39] could be thought of as a CLP system that uses bottom-up evaluation instead of Prolog-style search.

CHC solvers allow an additional type of clause beyond CLP clauses: *goal clauses*, where the head is a constraint [15, 34, 38]. The objective of CHC solving is to find a symbolic model to the predicates in the clauses that makes the overall system of constraints true; thus, goal clauses act as assertions that must be met. Consider this example:

```
p(0).
p(X + 1) :- p(X), X < 5.
X < 6 :- p(X).
```

The last clause is a goal clause. The overall system of constraints is satisfiable, and a CHC solver might assign the predicate `p` the symbolic solution $\mathtt{p}(x) \equiv 0 \leq x \leq 5$. CHC solving is a powerful tool; however, it does not seem necessary to encode the analyses that Formulog targets.[8]

The key difference between these paradigms and Formulog lies in the distinction between constraints and formulas. In CLP and CHC solving, evaluation is constrained by the satisfiability of the constraints that have been encountered. In contrast, formulas in Formulog are terms that can constructed and manipulated without being interpreted as constraints; they are only treated specially within functions like `is_sat`. For example, Formulog can naturally talk about 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 [21]. One could imagine extending Formulog's SMT interface to include a function `interpolate` that takes two formulas and returns a third (optional) formula, the interpolant; however, it is not clear how this would be possible in one of the constraint-based paradigms.

These different approaches are reflected in the typical program analysis use cases for Formulog versus CLP and CHC. In the case studies we have shown in this paper, the rules encoding an analysis are independent of the particular input program that is being analyzed: The rules generically state the implementation of the analysis, and can be run on any input program. Many CLP and CHC-based analyses take a different approach: given a particular input program (or, more generally, a system), encode a model of it using Horn clauses [15, 24]. In this case, the rules depend on the program

[8]However, we could extend Formulog with an interface to a CHC solver; this would make it to possible to reason about formula terms as CHCs.

being analyzed, and the solutions of queries over these rules reveal properties of the model.[9] For example, the SeaHorn verification framework checks programs by solving a CHC representation of their verification conditions [34].

While still supporting formula-based programming, it might be possible to design or implement Formulog in a way that more closely integrates Horn clause evaluation and reasoning about formula terms. This could make it easier to take advantage of possible synergies arising between the two. However, there are good arguments for keeping them separate. First, our approach enables Formulog to take advantage of independent improvements in Datalog evaluation and SMT solving; for example, this makes it possible to more easily apply Datalog optimizations to the Formulog runtime, such as automatic parallelization or incremental evaluation [64]. Second, our approach explicitly separates "easy" constraint solving (the Horn clauses) from "difficult" constraint solving (the SMT formulas), so that sophisticated and potentially expensive constraint solving mechanisms are used only where they are necessary.

## 6 Other related work

***Datalog-based analysis frameworks*** Subsequent to the early use of Datalog by Reps [58] for deriving demand-driven versions of interprocedural analyses, a variety of static analysis frameworks have been developed based on more-or-less standard Datalog, such as bddbddb [72], Chord [50], Doop [18], QL [8], and Soufflé [60]. There has also been work on automatically refining the abstractions used in Datalog analyses [74] and in example-based synthesis of analysis implementations in Datalog [2].

Flix [52] and IncA [64] 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 functions written in a pure functional language. Formulog would benefit from some form of recursive aggregation, and one of these lattice-based approaches might be a good fit. The paper introducing Datafun, a language combining Datalog and higher-order functional programming, uses dataflow analysis as a case study [7]. It might be possible to encode something like Formulog in Datafun; however, although it has recently been shown that Datafun can be evaluated using seminaive evaluation [6], it is not clear to what extent other Datalog optimizations can be applied to Datafun programs. By interweaving Datalog with functional programming, Formulog, Flix, and Datafun are related to functional logic programming [3]. However, the functional fragment of Formulog is less expressive than what is typically found in such languages, as functions are not first-class values and not higher-order.

[9] A notable exception is ARMC, an abstraction refinement-based model checker written in a CLP language [56]. This implementation relies heavily on Prolog's non-logical features (like `assert`).

Formulog is closest in spirit to Calypso [1, 35], a Datalog variant that has an interface to a SAT solver and specialized support for bottom-up analyses. Formulas in Calypso are opaque terms created through predicates; they cannot be directly inspected. Formulog's approach to constructing and manipulating formulas, via its ML fragment, is a better fit for the added complexity of SMT formulas (relative to boolean formulas, which is what Calypso supports).

***Other logic programming languages*** The higher-order logic programming language $\lambda$Prolog provides a natural way to represent logical formulas using a form of higher-order abstract syntax based on $\lambda$-terms and higher-order unification [54, 55]. Although this representation of formulas simplifies some aspects of using formulas (e.g., correctly handling binders), 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 [19, 30]. Common extensions allow users to easily specify constraints on the shape of the stable model that will be found. For example, many ASP systems support cardinality constraints of the form $l\ \{L_1, \dots, L_n\}\ k$; such an atom is true if between $l$ and $k$ of the literals $L_1$ through $L_n$ are true. ASP enables concise encoding of classic NP-complete constraint problems such as graph $k$-coloring, but it is not obviously applicable to the static analysis problems we are interested in.

***Type system engineering*** PLT Redex [26] and Spoofax [44] offer support for 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 [69]. 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$^{Z3}$ [46] supports mixed computations combining normal Scala evaluation and Z3 solving; Formulog intentionally avoids this level of integration. Smten [68] is a solver-aided language that supports both concrete and symbolic evaluation; Rosette [66] is a framework for creating solver-aided languages that have this property.

## 7 Conclusion

Formulog is a variant of Datalog that supports constructing, manipulating, and reasoning about logical terms via an interface to an SMT solver. Our case studies show that diverse formula-based static analyses – refinement type checking, bottom-up points-to analysis, and symbolic evaluation – can be concisely encoded in Formulog, and that Datalog optimizations like automatic parallelization and goal-directed program rewriting can be advantageously applied to them.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

## References

[1] 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.

[2] 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.

[3] Sergio Antoy and Michael Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (2010), 74–85.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa.

[11] Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *The Journal of Logic Programming* 10, 3-4 (1991), 255–299.

[12] João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. 2011. Polymorphic Contracts. In *Proceedings of the 20th European Symposium on Programming*. 18–37.

[13] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 193–207.

[14] 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.

[15] 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.

[16] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 169–190.

[17] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception Analysis and Points-to Analysis: Better Together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*. 1–12.

[18] 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.

[19] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. 2011. Answer Set Programming at a Glance. *Commun. ACM* 54, 12 (2011), 92–103.

[20] 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.

[21] 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.

[22] Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 207–212.

[23] 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.

[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.

[25] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (2005), 365–473.

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

[27] 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.

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

[29] Marco Gavanelli and Francesca Rossi. 2010. Constraint Logic Programming. In *A 25-Year Perspective on Logic Programming*. Springer, 64–86.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.
[35] Brian Hackett. 2010. *Type Safety in the Linux Kernel.* Ph.D. Dissertation. Stanford University.
[36] Brian Hackett and Alex Aiken. 2006. How is Aliasing Used in Systems Software?. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 69–80.
[37] Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60.
[38] 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.
[39] Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. 2011. $\mu$Z–An Efficient Engine for Fixed Points with Constraints. In *International Conference on Computer Aided Verification.* 457–462.
[40] 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.
[41] Joxan Jaffar and Michael J. Maher. 1994. Constraint Logic Programming: A Survey. *The Journal of Logic Programming* 19 (1994), 503–581.
[42] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *International Conference on Computer Aided Verification.* 422–430.
[43] 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.
[44] 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.
[45] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.
[46] 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.
[47] Robert Kowalski. 1979. Algorithm = Logic + Control. *Commun. ACM* 22, 7 (1979), 424–436.
[48] 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.
[49] 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.
[50] Percy Liang and Mayur Naik. 2011. Scaling Abstraction Refinement via Pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* 590–601.
[51] 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.
[52] 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.
[53] Michael Meskes and Jörg Noack. 1993. The Generalized Supplementary Magic-Sets Transformation for Stratified Datalog. *Inform. Process. Lett.* 47, 1 (1993), 31–41.
[54] 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.
[55] 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.
[56] 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.
[57] 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.
[58] 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.
[59] 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.
[60] 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.
[61] Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. *ACM Trans. Program. Lang. and Syst.* 39, 1 (2017), 3:1–3:36.
[62] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded.* Springer, 245–251.
[63] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *Proceedings of the VLDB Endowment* 12, 2 (2018), 141–153.
[64] 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.
[65] John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *Proceedings of the 2nd Summit on Advances in Programming Languages.* 18:1–18:14.
[66] 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.
[67] 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.
[68] Richard Uhler and Nirav Dave. 2013. Smten: Automatic Translation of High-Level Symbolic Computations into SMT Queries. In *International Conference on Computer Aided Verification.* 678–683.
[69] 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.
[70] 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.
[71] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming.* 269–282.
[72] 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.
[73] 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.

[74] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 239–248.

| | | | |
|---|---|---|---|
| **Types** | | | |
| Types | $\tau$ | ::= | $t \mid t\ \mathsf{smt} \mid t\ \mathsf{sym}$ |
| Pre-types | $t$ | ::= | $B \mid D\ \vec{\tau} \mid \alpha$ |
| Base types | $B$ | ::= | $\mathsf{bool} \mid \mathsf{bv}[\mathsf{k}]_{k \in \mathbb{N}^+} \mid \dots$ |
| **Contexts** | | | |
| Datatype declarations | $\Delta$ | ::= | $\cdot \mid \Delta, D : \forall \vec{\alpha}_i.\ \{\overrightarrow{c_j : \vec{\tau}_k}\}$ |
| Program declarations | $\Phi$ | ::= | $\cdot \mid \Phi, f : \forall \vec{\alpha}, \vec{\tau} \to \tau \mid \Phi, uf : \vec{t} \to t \mid \Phi, p \subseteq \vec{\tau}$ |
| Variable contexts | $\Gamma$ | ::= | $\cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha$ |
| **Terms** | | | |
| Programs | prog | ::= | $\vec{F}_i\ \vec{H}_j$ |
| Functions | $F$ | ::= | $\mathsf{fun}\ f(\vec{X}_i : \vec{\tau}_i) : \tau = e$ |
| Horn clauses | $H$ | ::= | $p(\vec{X}_i) :- \vec{P}_j$ |
| Premises | $P$ | ::= | $A \mid \neg A$ |
| Atoms | $A$ | ::= | $p(\vec{e}_i) \mid X = e$ |
| Expressions | $e$ | ::= | $c(\vec{e}_i) \mid k \mid X \mid f(\vec{e}_i) \mid p(\vec{e}_i) \mid \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2$ |
| | | | $\mid$ \`$\phi$\` $\mid \mathsf{match}\ e\ \mathsf{with}\ \overrightarrow{c_i(\vec{X}_j) \to e_i} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ |
| SMT formulas | $\phi$ | ::= | $x{:}t \mid c(\vec{\phi}_i) \mid uf(\vec{\phi}_i) \mid \mathsf{let}_\phi\ \phi_1 = \phi_2\ \mathsf{in}\ \phi_3 \mid \forall \phi_1.\ \phi_2 \mid ,e$ |
| Constants | $k$ | ::= | $\mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \dots$ |
| **Namespaces** | | | |
| Type modes | $m$ | ::= | $\mathsf{exp} \mid \mathsf{smt}$ |
| Datatype names | $D$ | $\in$ | ADTVar |
| Type variables | $\alpha$ | $\in$ | TVar |
| Constructors | $c$ | $\in$ | CtorVar |
| Formulog variables | $X$ | $\in$ | Var |
| SMT variables | $x$ | $\in$ | SMTVar |
| Predicates | $p$ | $\in$ | PredVar |

**Figure 13.** Syntax of Formulog's formal model

## A Formulog's formal model

We define a 'middleweight' formal model of Formulog, designing a type system (Section B) and an operational semantics (Section C), relating the two in a proof of type safety (Section D).

Our model characterizes Formulog as a two-level system (Figure 13), comprising Datalog-esque Horn clauses $H$ and first-order functions $F$; Horn clause "rules" are made up of premises $P$, where each premise is a series of (possibly negated) atoms $A$. Each atom $A$ either references a Datalog predicate or binds a variable to an expression $e$. Expressions themselves have two mutually recursive modes: ordinary functional computation $e$ and quoted SMT terms \`$\phi$\`, which can include unquoted expressions $,e$.

The Datalog fragment of Formulog is fairly standard syntactically, up to the addition of the atomic form $X = e$. We constrain premises to a sort of administrative normal form: predicate references apply only to variables, written $p(\vec{X}_i)$, and expression constraints bind variables, as in $Y = e$. Our implementation can handle compound premises like $p(e_1, e_2)$; our formal model would require rewriting such a premise to three premises: $p(X, Y)$, $X = e_1$, and $Y = e_2$ (for some fresh $X$ and $Y$).

The functional programming fragment fully annotates the types on its functions $F$; variable names in both fragments are written in capital letters. (Our implementation merely demands that the first letter be capitalized.) SMT variables are written in lowercase letters and annotated with their type, as in $x{:}t$. Code in the functional fragment can treat Datalog relations as predicates, i.e., $p(\vec{v}_i)$ returns true when $\vec{v}_i \in p$. In our implementation, some elements of $\vec{v}_i$ can be the wildcard ??, turning a Datalog predicate into a list. For example, if $p \subseteq \mathsf{bool} \times \mathsf{bv}[32]$, then: $p(\mathsf{true}, 42)$ yields a bool; $p(??, 42)$ returns a list of bools $b$ such that $p(b, 42)$ holds; $p(\mathsf{true}, ??)$ returns a list of bv[32]s $n$ such that $p(\mathsf{true}, n)$ holds; $p(??, ??)$ returns a list of $\mathsf{bool} \times \mathsf{bv}[32]$, i.e., the relation $p$. We don't include this behavior in our formal model.

**Type and typing context well formedness** $\boxed{\Delta \vdash \Gamma}$ $\boxed{\Delta; \Gamma \vdash_m \tau}$

$$\frac{}{\vdash \cdot}\ \Gamma\text{-Empty} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash_{\text{exp}} \tau}{\vdash \Gamma, x : \tau}\ \Gamma\text{-Var} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \alpha}\ \Gamma\text{-TVar}$$

$$\frac{}{\Gamma \vdash_m B}\ t\text{-Base} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash_m \alpha}\ t\text{-TVar} \qquad \frac{\Delta(D) = \forall \vec{\alpha}_i, \{\dots\} \quad \Gamma \vdash_m \tau_i}{\Gamma \vdash_m D\ \vec{\tau}_i}\ t\text{-ADT}$$

$$\frac{\Gamma \vdash_{\text{smt}} t}{\Gamma \vdash_{\text{exp}} t\ \mathsf{smt}}\ \tau\text{-SMT} \qquad \frac{\Gamma \vdash_{\text{smt}} t}{\Gamma \vdash_{\text{exp}} t\ \mathsf{sym}}\ \tau\text{-Sym}$$

**Datatype and program signature well formedness** $\boxed{\vdash \Delta}$ $\boxed{\vdash \Phi}$

$$\begin{aligned} \vdash \Delta \quad \Leftrightarrow \quad & \forall D : \forall \vec{\alpha}.\ \{c_1 : \vec{\tau}_1, \dots, c_n : \vec{\tau}_n\} \in \Delta \forall i, \\ & (1)\ \forall D' \in \text{dom}(\Delta),\ c_i \in \Delta(D') \Rightarrow D = D' \\ & (2)\ \vec{\alpha} \vdash_{\text{exp}} \tau_i \\ & (3)\ \forall \beta \in \vec{\alpha}, \beta \in \vec{\tau}_i \end{aligned}$$

$$\frac{}{\vdash \cdot}\ \Phi\text{-Empty} \qquad \frac{\vdash \Phi \quad \forall \beta \in \vec{\alpha}_i,\ \beta \in \vec{\tau}_j, \tau \quad \vec{\alpha}_i \vdash_{\text{exp}} \tau_j \quad \vec{\alpha}_i \vdash_{\text{exp}} \tau}{\vdash \Phi, f : \forall \vec{\alpha}_i, \vec{\tau}_j \to \tau}\ \Phi\text{-Fun}$$

$$\frac{\vdash \Phi \quad \cdot \vdash_{\text{exp}} \tau_i}{\vdash \Phi, p \subseteq \vec{\tau}_i}\ \Phi\text{-Rel} \qquad \frac{\vdash \Phi \quad \cdot \vdash_{\text{smt}} t_i \quad \cdot \vdash_{\text{smt}} t}{\vdash \Phi, \mathit{uf} : \vec{t}_i \to t}\ \Phi\text{-UFun}$$

**Program and function typing** $\boxed{\Delta; \Phi \vdash \text{prog}}$

$$\frac{\vdash \Delta \quad \vdash \Phi \quad \Delta; \Phi \vdash F_i \quad \Delta; \Phi \vdash H_j}{\Delta; \Phi \vdash \vec{F}_i\ \vec{H}_j}\ \text{prog-WF}$$

**Figure 14.** Type, context, and definition well formedness; top-level program typing

The set of available base types $B$ must include bool at a minimum; any other SMT-embeddable base types are acceptable, e.g., $k$-width bit vectors for a statically known $k$.

As a matter of notation, we write $\vec{e}_i$ for a metavariable $e$ to mean a possibly empty sequence of $e$s, indexed by $i$. When more than one variable shares the same index, we mean that those sequences must be of the same length (e.g., in the match syntax, each branch of a match is a triple of a constructor $c$, a vector of variable names for $c$'s arguments, and a corresponding single expression).

## B Formulog's type system

We begin by presenting type checking rules for Formulog (Figures 14, 15, 16, and 17). Our *implementation* of Formulog not only performs type checking, but can also perform type inference, e.g., automatically finding type variable substitutions.

Our types are broken into two levels: types $\tau$ and pre-types $t$. Every pre-type $t$ can be directly considered as a type, but there are two additional types: $t$ smt, the type of SMT formulas yielding $t$, and $t$ sym, the type of SMT variables of type $t$. We factor the syntax in this way to prevent anomalies like bool smt smt, which would mean SMT formulas that yield SMT formulas that yield booleans. It is *not* the case, however, that every pre-type $t$ is necessarily representable as an SMT type, because datatypes may contain SMT formulas as arguments; we discuss how we categorize SMT-representable types shortly.

Before we begin, some further notational clarification. Rules are named by their primary subjects followed by a hyphen and a descriptive name. Whenever we use indices in rules, we will always map (stating a single premise in terms of the index, e.g., prog-WF) or fold (stating first, indexed, and last, e.g., $\vec{X}\vec{\tau}$-All) over the sequence. We omit the indices when selecting an element of a sequence or set (as in, e.g., $e$-Match in Figure 16).

All of our typing rules are in terms of a fixed set of datatype declarations $\Delta$ and program declarations $\Phi$ (Figure 13). Datatype declarations $\Delta$ map datatype names $D$ to some number of type arguments $\vec{\alpha}_i$ and a set of constructors $\vec{c}_j$, each of which takes some number of arguments of type $\vec{\tau}_k$; each $c_j$ can have a different number of arguments. Program declarations $\Phi$ collect the signatures of first-order polymorphic functions $f : \forall\vec{\alpha}, \vec{\tau} \to \tau$, uninterpreted functions for use in the SMT solver $uf : \vec{t} \to t$, and relations $p \subseteq \vec{\tau_i}$.

The highest level typing rule is prog-WF (Figure 14), which ensures that the declarations are well formed and each part of the program is well formed.

The context and type well formedness rules (Figure 14) are mostly straightforward, type well formedness being the most interesting. Each type can be found to be well formed in either SMT mode smt—i.e., it can be exported to the SMT solver–or in expression mode exp, meaning it cannot be. There is a sub-moding relationship: well formed types at smt are also well formed at exp, but not necessarily vice-versa: for example, there is no way to export an SMT formula or variable as the *object* of another SMT formula, only as a constituent. We assume that all Formulog constants are SMT representable, i.e., $\cdot \vdash_{\mathsf{smt}} \mathrm{typeof}(k)$ for all constants $k$. Datatype declarations are polymorphic, but we disallow phantom type variables. Datatypes can freely mutually recurse. Uninterpreted functions must be in terms of pre-types, and those pre-types must be closed and SMT representable (smt); functions and relations can use any well formed types (exp). Functions can be polymorphic but we disallow phantom type variables; relations have monomorphic types. Disallowing phantom types in constructors and functions and keeping relations monomorphic ensure that these forms are "reverse determinate", i.e., the types of their arguments uniquely determine their types.

Since the declaration environments $\Delta$ and $\Phi$ are statically determined for an entire program, we typically leave them *implicit*. Implicit parameters are in gray in the boxed rule schemata in the figures. In proofs we will treat these parameters explicitly, but we conserve space by stating the rules without threading implicit parameters through. For example, the datatype declarations $\Delta$ are necessary to ensure that $t$-ADT only allows us to name datatypes that have actually been defined. Rather than threading $\Delta$ through every rule for context and type well formedness, we write $\Delta$ in the rule schemata.

The type checking of the Datalog fragment of Formulog (Figure 15) must encode two Datalog invariants in addition to conventional typing constraints: the range restriction, i.e., every variable in the head of a rule appears somewhere in a premise; and appropriate binding, i.e., it is possible to interpret a Horn clause in such a way that all of the variables will be bound at the end. Our formal rules ensure that the program has correct binding structure for a left-to-right evaluation of each Horn clause. Our implementation can determine whether or not any ordering will work and will reorder programs into an appropriate order automatically. Our formal model does not enforce that negation of relations is appropriately stratified, though doing so would be easy: the relation-and-function call graph should not have any "negative" edge in a cycle, where a negative edge is created whenever there is a negated predicate in a rule body or a predicate is invoked as a function.

Concretely, $H$-Clause ensures that (a) a left-to-right binding order produces some appropriate final context $\Gamma'$ (via the premise typing judgment), (b) the range restriction is satisfied, ($\vec{X_i} \subseteq \Gamma'$) by making sure that (c) every variable is well typed and bound ($\Gamma' \vdash \vec{X_i}, \vec{\tau_i} \rhd \Gamma'$—having the same $\Gamma'$ means no new bindings were introduce when checking the head variables).

Premise typing $\Gamma \vdash P \rhd \Gamma$ and variable binding $\Gamma \vdash x, \tau \rhd \Gamma$ and typing work together to generate appropriate types for each premise. Positive references to relations are well formed in binding $\Gamma'$ according to $P$-PosAtom when (a) the use is well typed ($p \subseteq \vec{\tau_i} \in \Phi$) and (b) the variables used in the premise yield the binding $\Gamma'$. Negative references to relations $!p(\vec{X_i})$ additionally require that all of the $X_i$ be already bound, i.e., the resulting $\Gamma$ is the same as the starting one. We split expression equality constraints $Y = e$ into three cases:

1. $Y$ is bound and $e$ is a constructor $c(\vec{X_i})$ ($P$-EqCtor-BF)
2. $e$ is a constructor $c(\vec{X_i})$ and each of the $X_i$ are bound ($P$-EqCtor-FB)
3. $e$ is not a constructor ($P$-EqExpr)

In the case where $Y$ is bound and $e = c(\vec{X_i})$ with each $X_i$ bound, either of the $P$-EqCtor-... cases could apply. It is critical to avoid the case where both $Y$ and some of the $X_i$ are unbound, in which case we would need to perform higher-order unification.

The binding rules come in three forms: $X\tau$-Bind for adding a new binding, $X\tau$-Check for ensuring that an already bound variable is matched at appropriate type, and a vectorized form $\vec{X}\vec{\tau}$-All for folding over a sequence of such bindings. Note that the resulting bindings are the same, i.e., $\Gamma \vdash X, \tau \rhd \Gamma$, if and only if $X \in \mathrm{dom}(\Gamma)$; the same holds for vectors of variables and types, as well (Lemma E.5).

We split the rules for expressions $e$ and formulas $\phi$ in two parts (Figures 16 and 17, respectively). Expression typing is conventional for functional languages. We adopt a declarative style for type substitutions ($e$-Ctor, $e$-Fun, $e$-Match). Our actual implementation uses Hindley–Damas–Milner type inference [22, 37] to find the correct types to use. As to Formulog-specific features, we ensure type well formedness is in exp-mode; the `` `$\phi$` `` expression switches from expression mode to formula mode. Relations $p \subseteq \vec{\tau_i} \in \Phi$ are treated as if they are functions of type $\vec{\tau_i} \to \mathsf{bool}$. Since Datalog predicates can occur in functional terms, we must use the control-flow graph of the program when analyzing for stratification. Consider the following program:

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

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

$$\frac{\Gamma \vdash X_0, \tau_0 \rhd \Gamma_1 \quad \dots \quad \Gamma_i \vdash X_i, \tau_i \rhd \Gamma_{i+1} \quad \dots \quad \Gamma_n \vdash X_n, \tau_n \rhd \Gamma'}{\Gamma \vdash \vec{X_i}, \vec{\tau_i} \rhd \Gamma'}\ \vec{X}\vec{\tau}\text{-}\textsc{All}$$

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

$$\frac{p \subseteq \vec{\tau_i} \in \Phi \qquad \Gamma \vdash \vec{X_i}, \vec{\tau_i} \rhd \Gamma'}{\Gamma \vdash p(\vec{X_i}) \rhd \Gamma'}\ P\text{-}\textsc{PosAtom} \qquad \frac{p \subseteq \vec{\tau_i} \in \Phi \qquad \Gamma \vdash \vec{X_i}, \vec{\tau_i} \rhd \Gamma}{\Gamma \vdash !p(\vec{X_i}) \rhd \Gamma}\ P\text{-}\textsc{NegAtom}$$

$$\frac{\Delta(D) = \forall \vec{\alpha_j}, \{\dots, c : \vec{\tau_i}, \dots\} \qquad \Gamma \vdash Y, \tau[\vec{\tau_j'}/\vec{\alpha_j}] \rhd \Gamma \qquad \Gamma \vdash \vec{X_i}, \vec{\tau_i}[\vec{\tau_j'}/\vec{\alpha_j}] \rhd \Gamma'}{\Gamma \vdash Y = c(\vec{X_i}) \rhd \Gamma'}\ P\text{-}\textsc{EqCtor-BF}$$

$$\frac{\Delta(D) = \forall \vec{\alpha_j}, \{\dots, c : \vec{\tau_i}, \dots\} \qquad \Gamma \vdash Y, \tau[\vec{\tau_j'}/\vec{\alpha_j}] \rhd \Gamma' \qquad \Gamma \vdash \vec{X_i}, \vec{\tau_i}[\vec{\tau_j'}/\vec{\alpha_j}] \rhd \Gamma}{\Gamma \vdash Y = c(\vec{X_i}) \rhd \Gamma''}\ P\text{-}\textsc{EqCtor-FB}$$

$$\frac{e \neq c(\vec{X_i}) \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash Y, \tau \rhd \Gamma'}{\Gamma \vdash Y = e \rhd \Gamma'}\ P\text{-}\textsc{EqExpr}$$

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

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

**Figure 15.** Typing rules: Horn clauses (rules)

```
fun f(X : bv[32]) : bv[32] = if !p(X) then -X else X

input r(bv[32])
output p(bv[32])

p(Y) :- ... .

output q(bv[32], bv[32])

q(A, B) :- r(A), B = f(A).
```

Here the relation q calls f, which in turn relies on the negation of p. If p were to reference q, the circularity could lead to incorrect answers: it may be that a pair is missing from q at the time p not because said pair will never appear, but because we simply haven't computed it yet.

While Formulog's expressions *compute* values, the Formulog's formulas *construct* ASTs, to be shipped off to an SMT solver. Every $\phi$-… typing rule generates a value with an SMT type, i.e., either $t$ sym or $t$ smt for some $t$ that is well formed in smt-mode (Lemma D.9). SMT variables $x{:}t$ are written in lowercase to emphasize their distinction from expression variables $X$; these SMT variables will be used as names in the formulas sent to the SMT solver. We keep track of which terms are SMT variables $x{:}t$ of type $t$ sym (generated by $\phi$-Var) and which are plain SMT formulas of type $t$ smt (all other rules). We treat $t$ sym as a subtype of $t$ smt ($\phi$-Promote).

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

$$\frac{f : \forall \vec{\alpha}_j, \vec{\tau}_i \to \tau \in \Phi \qquad \vec{\alpha}_j, \overrightarrow{X_i : \tau_i} \vdash e : \tau}{\vdash \mathsf{fun}\ f(\vec{X}_i : \vec{\tau}_i) : \tau = e}\ F\text{-WF}$$

$$\frac{\vdash \Gamma \qquad \Gamma(X) = \tau}{\Gamma \vdash X : \tau}\ e\text{-Var} \qquad \frac{}{\Gamma \vdash k : \mathsf{typeof}(k)}\ e\text{-Const} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, X : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 : \tau_2}\ e\text{-Let}$$

$$\frac{\Delta(D) = \forall \vec{\alpha}_j, \{\ldots, c : \vec{\tau}_i, \ldots\} \qquad \Gamma \vdash_{\exp} \tau'_j \qquad \Gamma \vdash e_i : \tau_i[\tau'_j/\alpha_j]}{\Gamma \vdash c(\vec{e}_i) : D\ \vec{\tau}'_j}\ e\text{-Ctor} \qquad \frac{\Gamma \vdash \phi : \tau}{\Gamma \vdash \grave{}\phi\grave{} : \tau}\ e\text{-Quote}$$

$$\frac{p \subseteq \vec{\tau}_i \in \Phi \qquad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash p(\vec{e}_i) : \mathsf{bool}}\ e\text{-Rel} \qquad \frac{f : \forall \vec{\alpha}_j, \vec{\tau}_i \to \tau \in \Phi \qquad \Gamma \vdash_{\exp} \tau'_j \qquad \Gamma \vdash e_i : \tau_i[\tau'_j/\alpha_j]}{\Gamma \vdash f(\vec{e}_i) : \tau[\tau'_j/\alpha_j]}\ e\text{-Fun}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}\ e\text{-If}$$

$$\frac{\Gamma \vdash e : D\ \vec{\tau}_j \qquad \Delta(D) = \forall \vec{\alpha}_j, \{\ldots, c_i : \vec{\tau}_k, \ldots\} \qquad \Gamma, \overrightarrow{X_k : \tau_k[\tau_j/\alpha_j]} \vdash e_i : \tau}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \overrightarrow{c_i(\vec{X}_k) \to e_i} : \tau}\ e\text{-Match}$$

**Figure 16.** Typing rules: expressions; implicit parameters are in gray

The $\grave{}\phi\grave{}$ operator is an expression term that introduces a quoted SMT formulas; the $,e$ operator is the corresponding 'unquote' operator that introduces an expression ($\phi$-Unquote).

While unquoting generally suffices for embedding the results of expressions in formulas, we treat constructors specially so that we can mix concrete and symbolic (i.e., SMT) arguments in a single datatype constructor ($\phi$-Ctor). The SMT type equivalence $\equiv$ conflates the types $t$, $t$ sym, and $t$ smt, allowing us to write terms like:

$$\mathsf{let}\ H = 5\ \mathsf{in}\ \grave{}\mathsf{cons}(,H, l{:}(\mathsf{bv}[32]\ \mathsf{list}))\grave{}$$

Note that $H$ is an expression variable and $l$ is an SMT variable; our typing equivalence lets us mix them in the same list of 32-bit numbers. The $t$ sym type is used in $\phi$-Let and $\phi$-Forall, which construct SMT formuale that use binders. The only way to get a value of type $t$ sym is either with $\phi$-Var or with $\phi$-Unquote, as in $\mathsf{let}\ X = \grave{}x{:}\mathsf{bv}[32]\grave{}\ \mathsf{in}\ \grave{},x\grave{}$. The toSMT metafunction alters the type of $e$ to make sure it is SMT representable; toSMT relies on an erase function to avoid nesting $\ldots$ smt and $\ldots$ sym type constructors.

Uninterpreted functions must be applied to appropriate SMT types ($\phi$-UFun); recall that $\Phi$-UFun ensures that each uninterpreted function's types are SMT representable.

Finally, there are a suite of SMT constructors of the form $c^{\mathsf{SMT}}_{\ldots}$. Each of these special $c^{\mathsf{SMT}}_{\ldots}$ constructors is treated as an ordinary constructor by the operational semantics, even though the constructors don't appear in $\Delta$. We need these constructors to prove our type safety lemma for expressions and formulas (Lemma E.2). Each of these forms corresponds directly to a corresponding rule for constructing formulas, e.g., $\phi$-Var and $\phi$-SMT-Var.

Our formal model elides some of the detail of our SMT encoding, such as constructors for SMT operations like bitvector manipulation or equality. These operations are all encoded as SMT-specific constructors, i.e., particular $c^{\mathsf{SMT}}_{\mathsf{ctor}}(c, \ldots)$. There are some subtle issues around polymorphism and determinacy. Our implementation treats equality and other polymorphic SMT operations specially, where each use of a polymorphic operator must be fully instantiated. In practice our implementation can usually infer the instantiation; users must annotate in those places we cannot infer.

## C Operational semantics

Formulog's operational semantics operates over *worlds* $\mathcal{W}$ and substitutions $\theta$ (Figure 18); the semantics is a mix of small-step rules modeling a single application of a Datalog rule (Figure 19), which depend on a small-step rules explaining how premises unify (Figures 20 and 21); the premise semantics in turn depends on a semantics of expressions (Figures 22 and 23) and formulas (Figure 24).

**Formula well formedness** $\boxed{\Delta;\Phi;\Gamma \vdash \phi : \tau}$

$$\frac{\Gamma \vdash_{\mathsf{smt}} t}{\Gamma \vdash x{:}t : t\ \mathsf{sym}}\ \phi\text{-}\textsc{Var} \qquad \frac{\Gamma \vdash \phi : t\ \mathsf{sym}}{\Gamma \vdash \phi : t\ \mathsf{smt}}\ \phi\text{-}\textsc{Promote} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash {,}e : \mathrm{toSMT}(\tau)}\ \phi\text{-}\textsc{Unquote}$$

$$\frac{\Gamma \vdash \phi_X : t_1\ \mathsf{sym} \qquad \Gamma \vdash \phi_1 : t_1\ \mathsf{smt} \qquad \Gamma \vdash \phi_2 : t_2\ \mathsf{smt}}{\Gamma \vdash \mathsf{let}_\phi\ \phi_X = \phi_1\ \mathsf{in}\ \phi_2 : t_2\ \mathsf{smt}}\ \phi\text{-}\textsc{Let}$$

$$\frac{\Delta(D) = \forall \vec{\alpha_j}, \{\dots, c : \vec{\tau_i}, \dots\} \qquad \Gamma \vdash \phi_i : t_i'\ \mathsf{smt} \qquad \tau_i[t_j'/\alpha_j] \equiv t_i'\ \mathsf{smt} \qquad \Gamma \vdash_{\mathsf{smt}} t_j'}{\Gamma \vdash c(\vec{\phi_i}) : \mathrm{toSMT}(D\ \vec{t_j'})}\ \phi\text{-}\textsc{Ctor}$$

$$\frac{uf : \vec{t_i} \to t \in \Phi \qquad \Gamma \vdash \phi_i : t_i\ \mathsf{smt}}{\Gamma \vdash uf(\vec{\phi_i}) : t\ \mathsf{smt}}\ \phi\text{-}\textsc{UFun} \qquad \frac{\Gamma \vdash \phi_1 : t\ \mathsf{sym} \qquad \Gamma \vdash \phi_2 : \mathsf{bool\ smt}}{\Gamma \vdash \forall \phi_1.\ \phi_2 : \mathsf{bool\ smt}}\ \phi\text{-}\textsc{Forall}$$

$$\frac{\Gamma \vdash_{\mathsf{smt}} t}{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{var}}(x, t) : t\ \mathsf{sym}}\ \phi\text{-SMT-}\textsc{Var} \qquad \frac{}{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{const}}(k) : \mathrm{typeof}(k)\ \mathsf{smt}}\ \phi\text{-SMT-}\textsc{Const}$$

$$\frac{\Gamma \vdash v_1 : t_1\ \mathsf{sym} \qquad \Gamma \vdash v_2 : t_1\ \mathsf{smt} \qquad \Gamma \vdash v_3 : t_2\ \mathsf{smt}}{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{let}}(v_1, v_2, v_3) : t_2\ \mathsf{smt}}\ \phi\text{-SMT-}\textsc{Let}$$

$$\frac{\Delta(D) = \forall \vec{\alpha_j}, \{\dots, c : \vec{\tau_i}, \dots\} \qquad \Gamma \vdash v_i : t_i'\ \mathsf{smt} \qquad \tau_i[t_j'/\alpha_j] \equiv t_i'\ \mathsf{smt} \qquad \Gamma \vdash_{\mathsf{smt}} t_j'}{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{ctor}}(c, \vec{v_i}) : D\ \vec{t_j'}\ \mathsf{smt}}\ \phi\text{-SMT-}\textsc{Ctor}$$

$$\frac{\Gamma \vdash v_1 : t_1\ \mathsf{sym} \qquad \Gamma \vdash v_2 : \mathsf{bool\ smt}}{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{forall}}(v_1, v_2) : \mathsf{bool\ smt}}\ \phi\text{-SMT-}\textsc{Forall} \qquad \frac{uf : \vec{t_i} \to t \in \Phi \qquad \Gamma \vdash v_i : t_i\ \mathsf{smt}}{\Gamma \vdash c^{\mathsf{SMT}}_{\mathsf{uf}}(uf, \vec{v_i}) : t\ \mathsf{smt}}\ \phi\text{-SMT-}\textsc{UFun}$$

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

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

**SMT type equivalence** $\boxed{\tau \equiv \tau}$

$$\frac{}{B \equiv B}\ \equiv\text{-B} \qquad \frac{}{\alpha \equiv \alpha}\ \equiv\text{-}\textsc{TVar} \qquad \frac{\tau_i \equiv \tau_i'}{D\ \vec{\tau_i} \equiv D\ \vec{\tau_i'}}\ \equiv\text{-D} \qquad \frac{}{t \equiv t\ \mathsf{smt}}\ \equiv\text{-SMT}$$

$$\frac{}{t \equiv t\ \mathsf{sym}}\ \equiv\text{-SMT}\textsc{Sym} \qquad \frac{t_2 \equiv t_1}{t_1 \equiv t_2}\ \equiv\text{-}\textsc{Sym} \qquad \frac{t_1 \equiv t_2 \qquad t_2 \equiv t_3}{t_1 \equiv t_3}\ \equiv\text{-}\textsc{Trans}$$

**Figure 17.** Typing rules: formulas, SMT representations, and SMT type equivalence

Our worlds $\mathcal{W}$ are (subsets of) Herbrand models. Our small-step semantics iteratively builds up a world that is in fact a Herbrand model of the original relations in the program. We could have modeled our semi-naive evaluation model for Formulog in more detail, showing that all programs generate a world $\mathcal{W}$ that is a well typed Herbrand model of the user's program (possibly taking infinite time to do so). Doing so wouldn't add anything materially interesting to our formulation.

Throughout, the type system's goal is prevent a program yielding $\bot$, the bottom "wrong" value. Such a value denotes a serious, unrecoverable error, such as using a relation with the wrong arity or conditioning on a non-boolean. It is important to distinguish bad, $\bot$-yielding programs from those that simply fail to step. The goal of Datalog evaluation is to reach a fixpoint, i.e., to be unable to step! Finally, as is common, we assume that built-in operations do not yield $\bot$, i.e., they are total. While we

**Namespaces**

| | | | |
|---|---|---|---|
| World | $\mathcal{W}$ | $\in$ | $\text{PSym} \to \mathcal{P}(\text{Val} \times \cdots \times \text{Val})$ |
| World or error | $\mathcal{W}_\perp$ | $\in$ | World + Error |
| Substitution | $\theta$ | $\in$ | $\text{Var} \rightharpoonup \text{Val}$ |
| Substitution or error | $\theta_\perp$ | $\in$ | Substitution + Error |

**Values**

| | | | |
|---|---|---|---|
| Results | $v_\perp$ | $::=$ | $v \mid \perp$ |
| Values | $v \in \text{Val}$ | $::=$ | $k \mid c(\vec{v}_i)$ |
| Unifiable term | $u \in \text{UTerm}$ | $::=$ | $X \mid k \mid c(\vec{u}_i)$ |

**Substitution and world well formedness** $\boxed{\Delta; \Phi; \Gamma \models \theta}$ $\boxed{\Delta; \Phi \models \mathcal{W}}$

$$\Gamma \models \theta \iff \forall X \in \text{dom}(\Gamma) \begin{cases} (1)\ X \in \text{dom}(\theta) \\ (2)\ \cdot \vdash \theta(X) : \Gamma(X) \end{cases}$$

$$\Delta; \Phi \models \mathcal{W} \iff \forall p \subseteq \vec{\tau}_i \in \Phi \begin{cases} (1)\ p \in \text{dom}(\mathcal{W}) \\ (2)\ \vec{v}_j \in \mathcal{W}(p) \Rightarrow i = j \\ (3)\ \forall \vec{v}_i \in \mathcal{W}(p),\ \Delta; \Phi; \cdot \vdash v_i : \tau_i \end{cases}$$

**Figure 18.** Definitions for semantics

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

$$\frac{\cdot \vdash P_0 \to \theta_1 \quad \ldots \quad \theta_i \vdash P_i \to \theta_{i+1} \quad \ldots \quad \theta_n \vdash P_n \to \theta}{\vec{F}; \mathcal{W} \vdash p(\vec{X}_j) :- \vec{P}_i \to \mathcal{W}[p \mapsto \mathcal{W}(p) \cup \theta(\vec{X}_j)]}\ \textsc{Clause}$$

$$\frac{\cdot \vdash P_0 \to \theta_1 \quad \ldots \quad \theta_i \vdash P_i \to \theta_{i+1} \quad \ldots \quad \theta_n \vdash P_n \to \perp}{\vec{F}; \mathcal{W} \vdash p(\vec{X}_j) :- \vec{P}_i \to \perp}\ \textsc{Clause-E1}$$

$$\frac{\cdot \vdash P_0 \to \theta_1 \quad \ldots \quad \theta_i \vdash P_i \to \theta_{i+1} \quad \ldots \quad \theta_n \vdash P_n \to \theta \qquad \vec{X}_j \not\subseteq \text{dom}(\theta)}{\vec{F}; \mathcal{W} \vdash p(\vec{X}_j) :- \vec{P}_i \to \perp}\ \textsc{Clause-E2}$$

**Figure 19.** Clause semantics

could in principle design a type system for Formulog that avoids, say, division by zero, we are more interested in making the hard parts easy (generating well typed SMT formulas) rather than making the easy parts foolproof (statically protecting partial functions).

Rules of the form . . . -E$n$ denote $\perp$-yielding rules. Each such rule characterizes a form of wrongness avoided by our static type system. We write $v_\perp$ to denote the disjoint sum of values $v$ and the wrong value $\perp$.

During correct execution, Clause takes a Horn clause $p(\vec{X}_j) :- \vec{P}_i$, executes each premise $P_i$ from left to right, yielding a final substitution for the variables $\vec{X}_j$ in the head of the rule. There are two possible failing rules. Clause-E1 simply propagates errors from premises; Clause-E2 fails because not every $X_j$ in the head of the rule is bound by the end. Since Formulog enforces the range restriction ($H$-Clause), Clause-E2 can never apply in a well typed program. Finally, the Clause* operational rules and the $H$-Clause typing rule both use the fixed, given order of premises for checking. Different orderings induce different binding orders, some of which may succeed and some of which may not. Our actual implementation is more clever and less rigid: it will reorder clauses to make a query plan with safe binding structure, if it exists.

The premise semantics (Figure 20) uses unification (Figure 21) to match and bind variables. Positive atoms $p(\vec{X}_i)$ try to unify their arguments with a tuple for $p$ drawn from the world $\mathcal{W}$ (PosAtom). Negative atoms $p(\vec{X}_i)$ require that all of their arguments $X_i$ are already bound (NegAtom); failing to find such bound terms yields an error (NegAtom-E). Rules for equations

**Premise semantics** $\boxed{\vec{F}; \mathcal{W}; \theta \vdash P \longrightarrow \theta_\perp}$

$$\frac{\vec{v} \in \mathcal{W}(p) \qquad \theta \vdash \vec{X} \sim \vec{v} : \theta'_\perp}{\mathcal{W}; \theta \vdash p(\vec{X}) \longrightarrow \theta'_\perp}\ \textsc{PosAtom} \qquad \frac{\theta(\vec{X}) = \vec{v} \qquad \vec{v} \notin \mathcal{W}(p)}{\mathcal{W}; \theta \vdash !p(\vec{X}) \longrightarrow \theta}\ \textsc{NegAtom}$$

$$\frac{\theta \vdash Y \sim c(\vec{X}) : \theta_\perp}{\mathcal{W}; \theta \vdash Y = c(\vec{X}) \longrightarrow \theta_\perp}\ \textsc{EqCtor} \qquad \frac{e \text{ is not a constructor} \qquad \mathcal{W}; \theta \vdash e \Downarrow_e v \qquad \theta \vdash Y \sim v : \theta'_\perp}{\mathcal{W}; \theta \vdash Y = e \longrightarrow \theta'_\perp}\ \textsc{EqExpr}$$

$$\frac{\vec{x} \not\subseteq \mathrm{dom}(\theta)}{\mathcal{W}; \theta \vdash !p(\vec{x}) \longrightarrow \perp}\ \textsc{NegAtom-E} \qquad \frac{e \text{ is not a constructor} \qquad \mathcal{W}; \theta \vdash e \Downarrow_e \perp}{\mathcal{W}; \theta \vdash Y = e \longrightarrow \perp}\ \textsc{EqExpr-E}$$

**Figure 20.** Premise semantics

**Value unification** $\boxed{\theta \vdash u \sim v \triangleright \theta}$ $\boxed{\theta \vdash \vec{u} \sim \vec{v} \triangleright \theta}$

$$\frac{\theta(X) = v}{\theta \vdash X \sim v \triangleright \theta}\ uv\text{-}\textsc{Eq-Var} \qquad \frac{X \notin \mathrm{dom}(\theta)}{\theta \vdash X \sim v \triangleright \theta[X \mapsto v]}\ uv\text{-}\textsc{Bind-Var}$$

$$\frac{}{\theta \vdash k \sim k \triangleright \theta}\ uv\text{-}\textsc{Constant} \qquad \frac{\theta \vdash \vec{u_i} \sim \vec{v_i} \triangleright \theta'}{\theta \vdash c(\vec{u_i}) \sim c(\vec{v_i}) \triangleright \theta'}\ uv\text{-}\textsc{Ctor}$$

$$\frac{\theta \vdash u_0 \sim v_0 \triangleright \theta_1 \qquad \ldots \qquad \theta_1 \vdash u_i \sim v_i \triangleright \theta_i \qquad \ldots \qquad \theta_n \vdash u_n \sim v_n \triangleright \theta'}{\theta \vdash \vec{u_i} \sim \vec{v_i} \triangleright \theta}\ \vec{u}\vec{v}\text{-}\textsc{All}$$

**Unification** $\boxed{\theta \vdash u \sim u : \theta_\perp}$ $\boxed{\theta \vdash \vec{u} \sim \vec{u} : \theta_\perp}$

$$\frac{\theta(u_1) = v_1 \qquad \theta(u_2) = v_2 \qquad v_1 = v_2}{\theta \vdash u_1 \sim u_2 : \theta}\ uu\text{-BB} \qquad \frac{\theta(u_1) \neq v_1 \qquad \theta(u_2) = v_2 \qquad \theta \vdash u_1 \sim v_2 \triangleright \theta'}{\theta \vdash u_1 \sim u_2 : \theta'}\ uu\text{-FB}$$

$$\frac{\theta(u_1) = v_1 \qquad \theta(u_2) \neq v_2 \qquad \theta \vdash u_2 \sim v_1 \triangleright \theta'}{\theta \vdash u_1 \sim u_2 : \theta'}\ uu\text{-BF} \qquad \frac{\theta(u_1) \neq v_1 \qquad \theta(u_2) \neq v_2}{\theta \vdash u_1 \sim u_2 : \perp}\ uu\text{-FF}$$

$$\frac{\theta \vdash u_i \sim u'_i \triangleright \theta_i}{\theta \vdash \vec{u_i} \sim \vec{u'_i} \triangleright \theta\vec{\theta_i}}\ \vec{u}\vec{u}\text{-}\textsc{All} \qquad \frac{\ldots \qquad \theta \vdash u \sim u' \triangleright \perp \qquad \ldots}{\theta \vdash \vec{u_i} \sim \vec{u'_i} \triangleright \perp}\ \vec{u}\vec{u}\text{-}\textsc{All-E}$$

$$\theta(c(\vec{u_i})) = c(\overrightarrow{\theta(u_i)})$$

**Figure 21.** Unification

also use unification, whether for a constructor over variables (EqCtor) or an expression (EqExpr). The latter can fail if evaluation fails (EqExpr-E). Before discussing term evaluation, we give rules for unification.

Unification is split into two levels. *Unification* proper takes a pair of unifiable terms—values with variables in them—and tries to yield a substitution. *Value unification* takes a unifiable term and a value and tries to yield a substitution. The unification rules are of the form *uu*-. . . . These rules analyze the two unifiable terms to find which side is completely bound—i.e., applying $\theta$ can completely fill in the variables—and so can be passed to value unification as a value. The B and F in these rules stand for Bound and Free. The only error in unification is in *uu*-FF, when neither unifiable term is bound to a value. We write a case for when both unifiable terms are bound (*uu*-BB) and require that they are directly equal—but it would also work to drop this rule and rely on value unification to identify the equality.

Value unification rules are of the form *uv*-. . . . The rules here lookup variables in the unifiable term and either check that the binding conforms to the given value (*uv*-Eq-Var, cf. $X\tau$-Check) or binds the value (*uv*-Bind-Var, cf. $X\tau$-Bind). The remaining value unification rules match the structure of the unifiable term to the structure of the value (*uv*-Constant, *uv*-Ctor) or fold

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

$$\frac{}{\mathcal{W}; \theta \vdash \cdot \Downarrow_{\vec{e}} \cdot}\ \Downarrow_{\vec{e}}\text{-EMPTY} \qquad \frac{\mathcal{W}; \theta \vdash e \Downarrow_e v \qquad \mathcal{W}; \theta \vdash \vec{e} \Downarrow_{\vec{e}} \vec{v}}{\mathcal{W}; \theta \vdash e, \vec{e} \Downarrow_{\vec{e}} v, \vec{v}}\ \Downarrow_{\vec{e}}\text{-ALL}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \vec{v}_i}{\mathcal{W}; \theta \vdash c(\vec{e}_i) \Downarrow_{\vec{e}} c(\vec{v}_i)}\ \Downarrow_e\text{-CTOR} \qquad \frac{\mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \bot}{\mathcal{W}; \theta \vdash c(\vec{e}_i) \Downarrow_e \bot}\ \Downarrow_e\text{-CTOR-E}$$

$$\frac{}{\mathcal{W}; \theta \vdash k \Downarrow_e k}\ \Downarrow_e\text{-CONST} \qquad \frac{\theta(X) = v}{\mathcal{W}; \theta \vdash X \Downarrow_e v}\ \Downarrow_e\text{-VAR} \qquad \frac{\mathcal{W}; \theta \vdash \phi \Downarrow_\phi v_\bot}{\mathcal{W}; \theta \vdash \grave{}\phi\grave{} \Downarrow_e v_\bot}\ \Downarrow_e\text{-QUOTE}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{e} \Downarrow_{\vec{e}} \vec{v} \qquad [\![\otimes]\!](\vec{v}) = v}{\mathcal{W}; \theta \vdash \otimes(\vec{e}) \Downarrow_e v}\ \Downarrow_e\text{-OP}$$

$$\frac{\text{fun } f(\vec{X}_i : \vec{\tau}_i) : \tau = e \in \vec{F} \qquad \mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \vec{v}_i \qquad \mathcal{W}; \theta[\vec{X}_i \mapsto \vec{v}_i] \vdash e \Downarrow_e v_\bot}{\mathcal{W}; \theta \vdash f(\vec{e}_i) \Downarrow_e v_\bot}\ \Downarrow_e\text{-FUN}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \vec{v}_i \qquad \vec{v}_i \in \mathcal{W}(p)}{\mathcal{W}; \theta \vdash p(\vec{e}_i) \Downarrow_e \text{true}}\ \Downarrow_e\text{-REL-T} \qquad \frac{\mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \vec{v}_i \qquad \vec{v}_i \notin \mathcal{W}(p)}{\mathcal{W}; \theta \vdash p(\vec{e}_i) \Downarrow_e \text{false}}\ \Downarrow_e\text{-REL-F}$$

$$\frac{\mathcal{W}; \theta \vdash e_1 \Downarrow_e v_1 \qquad \mathcal{W}; \theta[X \mapsto v_1] \vdash e_2 \Downarrow_e v_\bot}{\mathcal{W}; \theta \vdash \text{let } X = e_1 \text{ in } e_2 \Downarrow_e v_\bot}\ \Downarrow_e\text{-LET}$$

$$\frac{\mathcal{W}; \theta \vdash e \Downarrow_e c(\vec{v}_i \qquad \mathcal{W}; \theta[\overrightarrow{X_i \mapsto v_i}] \vdash e \Downarrow_e v_\bot}{\mathcal{W}; \theta \vdash \text{match } e \text{ with} \ldots c(\vec{X}_i) \to e \ldots \Downarrow_e v_\bot}\ \Downarrow_e\text{-MATCH}$$

$$\frac{\mathcal{W}; \theta \vdash e_1 \Downarrow_e \text{true} \qquad \mathcal{W}; \theta \vdash e_2 \Downarrow_e v_\bot}{\mathcal{W}; \theta \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_e v_\bot}\ \Downarrow_e\text{-ITET} \qquad \frac{\mathcal{W}; \theta \vdash e_1 \Downarrow_e \text{false} \qquad \mathcal{W}; \theta \vdash e_3 \Downarrow_e v_\bot}{\mathcal{W}; \theta \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_e v_\bot}\ \Downarrow_e\text{-ITEF}$$

**Figure 22.** Expression semantics

value unificaiton along a vector ($\vec{u}\vec{v}$-ALL). Value unification never produces $\bot$. It isn't an error when two values fail to unify, since one might have to search through many tuples for a relation in $\mathcal{W}$ to find a one that matches, say, a given constructor.

The expression semantics is an entirely conventional big-step semantics using explicit substitutions. The operational rules implicitly take the function definitions $\vec{F}$ for use in applications ($\Downarrow_e$-FUN).

There are a variety of wrong behaviors prevented by our type system, mostly concerning mismatches between values and elimination forms: unbound variables ($\Downarrow_e$-VAR-E); mistyped arguments to built-in operations ($\Downarrow_e$-OP-E2); function, relation and constructor arity errors ($\Downarrow_e$-FUN-E2, $\Downarrow_e$-REL-E2, $\Downarrow_e$-MATCH-E4); non-existent functions and relations ($\Downarrow_e$-FUN-E3, $\Downarrow_e$-REL-E3); conditionals on inappropriate values ($\Downarrow_e$-MATCH-E2, $\Downarrow_e$-ITE-E2); and ill formed constructor names ($\Downarrow_e$-MATCH-E3); The remaining rules propagate errors ($\Downarrow_e$-LET-E, $\Downarrow_e$-OP-E1, $\Downarrow_e$-FUN-E1, $\Downarrow_e$-REL-E1, $\Downarrow_e$-MATCH-E1, $\Downarrow_e$-ITE-E1). As mentioned in the early discussion of our semantics in this section, $\Downarrow_e$-OP-E2 is *not* about division by zero (a form of going wrong our type system *doesn't* prevent), but about mis-application of built-in functions, e.g., taking the boolean negation of a number.

The operational semantics on formulas is simple: the rules generate ASTs for the SMT solver using the $c^{\text{SMT}}_{\ldots}$ constructors: constants ($c^{\text{SMT}}_{\text{const}}$), SMT variables ($c^{\text{SMT}}_{\text{var}}$), SMT datatypes ($c^{\text{SMT}}_{\text{ctor}}$), let bindings ($c^{\text{SMT}}_{\text{let}}$), quantification ($c^{\text{SMT}}_{\text{forall}}$), and uninterpreted function application ($c^{\text{SMT}}_{\text{uf}}$). In order to keep ourselves honest, we add requirements to $\Downarrow_\phi$-LET and $\Downarrow_\phi$-FORALL that the $v_1$ must be of the form $c^{\text{SMT}}_{\text{var}}(x, t)$ for some $x$ and $t$, along with corresponding error rules. The use of $t$ sym in the corresponding typing rules ($\phi$-LET, $\phi$-FORALL) will guarantee this property, along with a simple notion of canonical forms (Lemma E.1).

When unquoting values resulting from evaluating expressions, we use the toSMT function to translate expression values into the SMT's AST. The toSMT function is an identity on SMT ASTs, but it explicitly tags the constants and Formulog-defined constructors using $c^{\text{SMT}}_{\text{const}}$ and $c^{\text{SMT}}_{\text{var}}$.

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

$$\frac{\mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \vec{v}_i \qquad \mathcal{W}; \theta \vdash e \Downarrow_e \bot}{\mathcal{W}; \theta \vdash \vec{e}_i, e, \vec{e}_j \Downarrow_{\vec{e}} \bot}\ \Downarrow_{\vec{e}}\text{-All-E}$$

$$\frac{X \notin \mathrm{dom}(\theta)}{\mathcal{W}; \theta \vdash X \Downarrow_e \bot}\ \Downarrow_e\text{-Var-E} \qquad \frac{\mathcal{W}; \theta \vdash e_1 \Downarrow_e \bot}{\mathcal{W}; \theta \vdash \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \Downarrow_e \bot}\ \Downarrow_e\text{-Let-E}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{e} \Downarrow_{\vec{e}} \bot}{\mathcal{W}; \theta \vdash \otimes(\vec{e}) \Downarrow_e \bot}\ \Downarrow_e\text{-Op-E1} \qquad \frac{\mathcal{W}; \theta \vdash \vec{e} \Downarrow_{\vec{e}} \vec{v} \qquad \vec{v} \notin \mathrm{dom}([\![\otimes]\!])}{\mathcal{W}; \theta \vdash \otimes(\vec{e}) \Downarrow_e \bot}\ \Downarrow_e\text{-Op-E2}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \bot}{\mathcal{W}; \theta \vdash f(\vec{e}_i) \Downarrow_e \bot}\ \Downarrow_e\text{-Fun-E1} \qquad \frac{\mathsf{fun}\ f(\vec{X}_i : \vec{\tau}_i) : \tau = e \in \vec{F} \qquad i \neq j}{\mathcal{W}; \theta \vdash f(\vec{e}_j) \Downarrow_e \bot}\ \Downarrow_e\text{-Fun-E2} \qquad \frac{f \notin \vec{F}}{\mathcal{W}; \theta \vdash f(\vec{e}_j) \Downarrow_e \bot}\ \Downarrow_e\text{-Fun-E3}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{e}_i \Downarrow_{\vec{e}} \bot}{\mathcal{W}; \theta \vdash p(\vec{e}_i) \Downarrow_e \bot}\ \Downarrow_e\text{-Rel-E1} \qquad \frac{\mathcal{W}(p) \subseteq \mathcal{P}(\overrightarrow{\mathrm{Val}_i}) \qquad i \neq j}{\mathcal{W}; \theta \vdash p(\vec{e}_j) \Downarrow_e \bot}\ \Downarrow_e\text{-Rel-E2} \qquad \frac{p \notin \mathrm{dom}(\mathcal{W})}{\mathcal{W}; \theta \vdash p(\vec{e}_j) \Downarrow_e \bot}\ \Downarrow_e\text{-Rel-E3}$$

$$\frac{\mathcal{W}; \theta \vdash e \Downarrow_e \bot}{\mathcal{W}; \theta \vdash \mathsf{match}\ e\ \mathsf{with}\ \overrightarrow{c_i(\vec{X}_j) \to e_i} \Downarrow_e \bot}\ \Downarrow_e\text{-Match-E1} \qquad \frac{\mathcal{W}; \theta \vdash e \Downarrow_e v \qquad v \neq c(\vec{v}')}{\mathcal{W}; \theta \vdash \mathsf{match}\ e\ \mathsf{with}\ \overrightarrow{c_i(\vec{X}_j) \to e_i} \Downarrow_e \bot}\ \Downarrow_e\text{-Match-E2}$$

$$\frac{\mathcal{W}; \theta \vdash e \Downarrow_e c(\vec{v}_k) \qquad c \notin \{\vec{c}_i\}}{\mathcal{W}; \theta \vdash \mathsf{match}\ e\ \mathsf{with}\ \overrightarrow{c_i(\vec{X}_j) \to e_i} \Downarrow_e \bot}\ \Downarrow_e\text{-Match-E3} \qquad \frac{\mathcal{W}; \theta \vdash e \Downarrow_e c(\vec{v}_k) \qquad j \neq k}{\mathcal{W}; \theta \vdash \mathsf{match}\ e\ \mathsf{with}\ \overrightarrow{\ldots c(\vec{X}_j \ldots) \to e_i} \Downarrow_e \bot}\ \Downarrow_e\text{-Match-E4}$$

$$\frac{\mathcal{W}; \theta \vdash e_1 \Downarrow_e \bot}{\mathcal{W}; \theta \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_e \bot}\ \Downarrow_e\text{-Ite-E1} \qquad \frac{\mathcal{W}; \theta \vdash e_1 \Downarrow_e v \qquad v \notin \{\mathsf{true}, \mathsf{false}\}}{\mathcal{W}; \theta \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_e \bot}\ \Downarrow_e\text{-Ite-E2}$$

**Figure 23.** Expression semantics (error rules)

## D Metatheory

We break the metatheory into two parts: lemmas characterizing the SMT conversion (Section D.1) and lemmas showing type safety (Section E). The SMT lemmas culminate in two proofs: first, regularity (Lemma D.9) guarantees that (a) every type or context generated by the operational semantics is well formed and (b) that formula evaluation generates well typed SMT ASTs; second, we show that SMT conversion of values agrees with SMT conversion of types (Lemma D.11). Type safety culminates in theorems showing that premises don't yield $\bot$ and generate well typed substitutions (Lemma E.4) and so Horn clauses (a) never yield $\bot$ (Theorem E.6) and (b) take well typed worlds to well typed worlds (Theorem E.7).

### D.1 SMT conversion

We show a variety of properties of the erasure and SMT conversion functions: erasures and SMT conversion are equivalent to their original types (Lemmas D.1 and D.2); erasures and SMT conversion yields well formed types (Lemmas D.3 and D.4); types well formed in smt-mode are well formed in exp-mode (Lemma D.5); weakening and strengthening of typing contexts (Lemmas D.6 and D.7); type variable substitution (Lemma D.8)—we have no need of a value substitution lemma because our semantics uses environments; regularity (Lemma D.9; and, finally, that SMT conversion of values agrees with SMT conversion of types (Lemma D.11).

**Lemma D.1** (Erasures are SMT-equivalent). *$\tau \equiv \mathrm{erase}(\tau)$*

*Proof.* By induction on $\tau$.

($\tau = B$) We have $\mathrm{erase}(B) = B$, so by $\equiv$-B.

($\tau = \alpha$) We have $\mathrm{erase}(\alpha) = \alpha$, so by $\equiv$-TVar.

($\tau = D\ \vec{\tau}_i$) We have $\mathrm{erase}(D\ \vec{\tau}_i) = D\ \overrightarrow{\mathrm{erase}(\tau_i)}$, so by $\equiv$-D and the IHs on each $\tau_i$.

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

$$\frac{}{\mathcal{W}; \theta \vdash \cdot \Downarrow_{\vec{\phi}} \cdot}\ \Downarrow_{\vec{\phi}}\text{-Empty} \qquad \frac{\mathcal{W}; \theta \vdash \phi \Downarrow_{\phi} v \qquad \mathcal{W}; \theta \vdash \vec{\phi}_i \Downarrow_{\vec{\phi}} \vec{v}_i}{\mathcal{W}; \theta \vdash \phi, \vec{\phi}_i \Downarrow_{\vec{\phi}} v, \vec{v}_i}\ \Downarrow_{\vec{\phi}}\text{-All}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{\phi}_i \Downarrow_{\vec{\phi}} \vec{v}_i \qquad \mathcal{W}; \theta \vdash \phi \Downarrow_{\phi} \perp}{\mathcal{W}; \theta \vdash \vec{\phi}_i, \phi, \vec{\phi}_j \Downarrow_{\vec{\phi}} \perp}\ \Downarrow_{\vec{\phi}}\text{-All-E} \qquad \frac{}{\mathcal{W}; \theta \vdash x{:}t \Downarrow_{\phi} c^{\mathsf{SMT}}_{\mathsf{var}}(x, t)}\ \Downarrow_{\phi}\text{-Var}$$

$$\frac{\mathcal{W}; \theta \vdash e \Downarrow_{e} v}{\mathcal{W}; \theta \vdash {,}e \Downarrow_{\phi} \mathsf{toSMT}(v)}\ \Downarrow_{\phi}\text{-Unquote} \qquad \frac{\mathcal{W}; \theta \vdash e \Downarrow_{e} \perp}{\mathcal{W}; \theta \vdash {,}e \Downarrow_{\phi} \perp}\ \Downarrow_{\phi}\text{-Unquote-E}$$

$$\frac{\mathcal{W}; \theta \vdash \phi_1, \phi_2, \phi_3 \Downarrow_{\vec{\phi}} v_1, v_2, v_3 \qquad v_1 = c^{\mathsf{SMT}}_{\mathsf{var}}(x, t)}{\mathcal{W}; \theta \vdash \mathsf{let}_{\phi}\ \phi_1 = \phi_2\ \mathsf{in}\ \phi_3 \Downarrow_{\phi} c^{\mathsf{SMT}}_{\mathsf{let}}(v_1, v_2, v_3)}\ \Downarrow_{\phi}\text{-Let}$$

$$\frac{\mathcal{W}; \theta \vdash \phi_1, \phi_2, \phi_3 \Downarrow_{\vec{\phi}} \perp}{\mathcal{W}; \theta \vdash \mathsf{let}_{\phi}\ \phi_1 = \phi_2\ \mathsf{in}\ \phi_3 \Downarrow_{\phi} \perp}\ \Downarrow_{\phi}\text{-Let-E1} \qquad \frac{\mathcal{W}; \theta \vdash \phi_1, \phi_2, \phi_3 \Downarrow_{\vec{\phi}} v_1, v_2, v_3 \qquad v_1 \neq c^{\mathsf{SMT}}_{\mathsf{var}}(x, t)}{\mathcal{W}; \theta \vdash \mathsf{let}_{\phi}\ \phi_1 = \phi_2\ \mathsf{in}\ \phi_3 \Downarrow_{\phi} \perp}\ \Downarrow_{\phi}\text{-Let-E2}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{\phi} \Downarrow_{\vec{\phi}} \vec{v} \qquad c \neq c^{\mathsf{SMT}}_{\dots}}{\mathcal{W}; \theta \vdash c(\vec{\phi}) \Downarrow_{\phi} \mathsf{toSMT}(c(\vec{v}))}\ \Downarrow_{\phi}\text{-Ctor} \qquad \frac{\mathcal{W}; \theta \vdash \vec{\phi} \Downarrow_{\vec{\phi}} \perp}{\mathcal{W}; \theta \vdash c(\vec{\phi}) \Downarrow_{\phi} \perp}\ \Downarrow_{\phi}\text{-Ctor-E}$$

$$\frac{\mathcal{W}; \theta \vdash \phi_1, \phi_2 \Downarrow_{\vec{\phi}} v_1, v_2 \qquad v_1 = c^{\mathsf{SMT}}_{\mathsf{var}}(x, t)}{\mathcal{W}; \theta \vdash \forall \phi_1.\ \phi_2 \Downarrow_{\phi} c^{\mathsf{SMT}}_{\mathsf{forall}}(v_1, v_2)}\ \Downarrow_{\phi}\text{-Forall}$$

$$\frac{\mathcal{W}; \theta \vdash \phi_1, \phi_2 \Downarrow_{\vec{\phi}} \perp}{\mathcal{W}; \theta \vdash \forall \phi_1.\ \phi_2 \Downarrow_{\phi} \perp}\ \Downarrow_{\phi}\text{-Forall-E1} \qquad \frac{\mathcal{W}; \theta \vdash \phi_1, \phi_2 \Downarrow_{\vec{\phi}} v_1, v_2 \qquad v_1 \neq c^{\mathsf{SMT}}_{\mathsf{var}}(x, t)}{\mathcal{W}; \theta \vdash \forall \phi_1.\ \phi_2 \Downarrow_{\phi} \perp}\ \Downarrow_{\phi}\text{-Forall-E2}$$

$$\frac{\mathcal{W}; \theta \vdash \vec{\phi}_i \Downarrow_{\vec{\phi}} \vec{v}_i}{\mathcal{W}; \theta \vdash uf(\vec{\phi}_i) \Downarrow_{\phi} c^{\mathsf{SMT}}_{\mathsf{uf}}(uf, \vec{v}_i)}\ \Downarrow_{\phi}\text{-UFun} \qquad \frac{\mathcal{W}; \theta \vdash \vec{\phi}_i \Downarrow_{\phi} \perp}{\mathcal{W}; \theta \vdash uf(\vec{\phi}_i) \Downarrow_{\phi} \perp}\ \Downarrow_{\phi}\text{-UFun-E}$$

$$\frac{}{\mathcal{W}; \theta \vdash c^{\mathsf{SMT}}_{\dots}(\vec{v}_i) \Downarrow_{\phi} c^{\mathsf{SMT}}_{\dots}(\vec{v}_i)}\ \Downarrow_{\phi}\text{-SMT-Ctor}$$

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

$$\begin{aligned}
\mathsf{toSMT}(k) &= c^{\mathsf{SMT}}_{\mathsf{const}}(k) & \mathsf{toSMT}(c^{\mathsf{SMT}}_{\mathsf{ctor}}(c, \vec{v}_i)) &= c^{\mathsf{SMT}}_{\mathsf{ctor}}(c, \vec{v}_i)\\
\mathsf{toSMT}(c(\vec{v}_i)) &= c^{\mathsf{SMT}}_{\mathsf{ctor}}(c, \overrightarrow{\mathsf{toSMT}(v_i)}) & \mathsf{toSMT}(c^{\mathsf{SMT}}_{\mathsf{let}}(v_1, v_2, v_3)) &= c^{\mathsf{SMT}}_{\mathsf{let}}(v_1, v_2, v_3)\\
\mathsf{toSMT}(c^{\mathsf{SMT}}_{\mathsf{const}}(k)) &= c^{\mathsf{SMT}}_{\mathsf{const}}(k) & \mathsf{toSMT}(c^{\mathsf{SMT}}_{\mathsf{forall}}(v_1, v_2)) &= c^{\mathsf{SMT}}_{\mathsf{forall}}(v_1, v_2)\\
\mathsf{toSMT}(c^{\mathsf{SMT}}_{\mathsf{var}}(x, t)) &= c^{\mathsf{SMT}}_{\mathsf{var}}(x, t) & \mathsf{toSMT}(c^{\mathsf{SMT}}_{\mathsf{uf}}(uf, \vec{v}_i)) &= c^{\mathsf{SMT}}_{\mathsf{uf}}(uf, \vec{v}_i)
\end{aligned}$$

**Figure 24.** Formula semantics

($\tau = t$ smt) We have erase($t$ smt) = erase($t$). By ≡-SMT, we know that $t \equiv t$ smt; we are done by the IH and ≡-Trans.

($\tau = t$ sym) We have erase($t$ sym) = erase($t$). By ≡-SMTSym, we know that $t \equiv t$ sym; we are done by the IH and ≡-Trans. □

**Lemma D.2** (SMT conversion is SMT-equivalent). $\tau \equiv \mathsf{toSMT}(\tau)$

*Proof.* By induction on $\tau$.

($\tau = B$) We have toSMT($B$) = $B$ smt, so by ≡-SMT.

($\tau = \alpha$) We have $\text{toSMT}(\alpha) = \alpha$ smt, so by ≡-SMT.

($\tau = D\ \vec{\tau_i}$) We have $\text{toSMT}(D\ \vec{\tau_i}) = \text{erase}(D\ \vec{\tau_i})$ smt, so by ≡-SMT, ≡-Trans, and Lemma D.1.

($\tau = t$ smt) We have $\text{toSMT}(t \text{ smt}) = \text{erase}(t)$ smt, so by *equiv*-SMT, ≡-Trans, and Lemma D.1.

($\tau = t$ sym) We have $\text{toSMT}(t \text{ sym}) = \text{erase}(t)$ sym, so by *equiv*-SMTSym, ≡-Trans, and Lemma D.1. □

**Lemma D.3** (Erasure is well formed). *If* $\Gamma \vdash_m \tau$ *then* $\Gamma \vdash_m \text{erase}(\tau)$.

*Proof.* By induction on the well formedness derivation.

($t$-Base) Immediate, since $\text{erase}(B) = B$.

($t$-TVar) Immediate, since $\text{erase}(\alpha) = \alpha$.

($t$-ADT) By the IH on each constituent of $D\ \vec{\tau_i}$, and then by $t$-ADT.

($\tau$-SMT) Since $\text{erase}(t \text{ smt}) = \text{erase}(t)$, by the IH on $\Gamma \vdash_{\text{smt}} t$.

($\tau$-Sym) Since $\text{erase}(t \text{ sym}) = \text{erase}(t)$, by the IH on $\Gamma \vdash_{\text{smt}} t$. □

**Lemma D.4** (SMT conversion is well formed). *If* $\Gamma \vdash_m \tau$ *then* $\text{toSMT}(\tau) = t$ sym *or* $t$ smt *such that* $\Gamma \vdash_{\text{smt}} t$ *(and so* $\Gamma \vdash_{\text{exp}} \text{toSMT}(\tau)$*).*

*Proof.* By induction on the well formedness derivation.

($t$-Base) $\text{toSMT}(B) = B$ smt, which is well formed by $\tau$-SMT and $t$-B.

($t$-TVar) $\text{toSMT}(\alpha) = \alpha$ smt, which is well formed by $\tau$-SMT and $t$-TVar.

($t$-ADT) We know that $\text{erase}(D\ \vec{\tau_i})$ is still well formed by Lemma D.3; then by $\tau$-SMT.

($\tau$-SMT) Since it must be that $\Gamma \vdash_{\text{smt}} t$, then $\text{erase}(t)$ is also well formed by Lemma D.3; then by $\tau$-SMT.

($\tau$-Sym) Since it must be that $\Gamma \vdash_{\text{smt}} t$, then $\text{erase}(t)$ is also well formed by Lemma D.3; then by $\tau$-Sym. □

We say a type $t$ is an "SMT type" when $\Gamma \vdash_{\text{smt}} t$; a type $\tau$ is an SMT type when it is equal to an SMT type $t$ or when it is of the form $t$ smt or $t$ sym. Note that toSMT always produces an SMT type.

**Lemma D.5** (Type mode subsumption). *If* $\Gamma \vdash_{\text{smt}} \tau$ *then* $\Gamma \vdash_{\text{exp}} \tau$.

*Proof.* By induction on the well formedness derivation.

($t$-Base) Immediate.

($t$-TVar) Immediate.

($t$-ADT) By the IH on each constituent of $D\ \vec{\tau_i}$.

($\tau$-SMT) Contradictory.

($\tau$-Sym) Contradictory. □

**Lemma D.6** (Weakening). *If* $\vdash \Gamma$ *and* $\vdash \Gamma'$ *and* $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$ *then:*

1. $\vdash \Gamma, \Gamma'$
2. *If* $\Gamma \vdash_m \tau$ *then* $\Gamma, \Gamma' \vdash_m \tau$*;*
3. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma, \Gamma' \vdash e : \tau$*; and*
4. *If* $\Gamma \vdash \phi : \tau$ *then* $\Gamma, \Gamma' \vdash \phi : \tau$*.*

*Proof.* By mutual induction on the derivations.

***Contexts***

($\Gamma$-Empty) We have $\Gamma' = \cdot$; immediate by assumption.

($\Gamma$-Var) We have $\Gamma' = \Gamma'', X : \tau$. By the IH on $\Gamma''$ and $\Gamma$-Var, finding $\Gamma, \Gamma'' \vdash_m \tau$ by part (2) of the IH.

($\Gamma$-TVar) We have $\Gamma' = \Gamma'', \alpha$. By the IH on $\Gamma''$ and $\Gamma$-TVar.

***Type well formedness***

($t$-Base) Immediate, by $t$-Base.

($t$-TVar) Since $\Gamma$ and $\Gamma'$ have disjoint domains, we know $\alpha \in \Gamma$—by $t$-TVar.

($t$-ADT) By the IH on each constituent of $D\ \vec{\tau_i}$, followed by $t$-ADT.

($\tau$-SMT) By the IH on $\Gamma \vdash_{\mathsf{smt}} t$ and then $\tau$-SMT.

($\tau$-Sym) By the IH on $\Gamma \vdash_{\mathsf{smt}} t$ and then $\tau$-Sym. □

***Expressions***

($e$-Var) Since the domains are disjoint, $(\Gamma, \Gamma')(X) = \tau$ and we can still find $e$-Var.

($e$-Const) Immediate, by $e$-Const.

($e$-Let) By the $e$-Let and the IH on $e_1$ and $e_2$, $\alpha$-renaming $X$ appropriately.

($e$-Ctor) By $e$-Ctor and the IH, using part (2) on $\tau_j'$ and part (3) on $e_i$.

($e$-Quote) By the part (4) of the IH.

($e$-Rel) By $e$-Rel and the IH on each $e_i$.

($e$-Fun) By $e$-Fun and the the IH, using part (2) on $\tau_j'$ and part (3) on $e_i$.

($e$-If) By $e$-If and the IH on each of the $e_i$.

($e$-Match) By $e$-Match and the IH on $e$ and each of the $e_i$, $\alpha$-renaming each $X_k$ appropriately.

***Formulae***

($\phi$-Var) By $\phi$-Var and part (2) of the IH.

($\phi$-Promote) By $\phi$-Promote and the IH.

($\phi$-Unquote) By $\phi$-Unquote and part (3) of the IH.

($\phi$-Let) By $\phi$-Let and the IH on each of the constituent $\phi$.

($\phi$-Ctor) By $\phi$-Ctor and the IH, using part (2) on $t_j'$ and part (4) on $\phi_i$, observing that the type equivalences are preserved.

($\phi$-UFun) By $\phi$-UFun and the IH on each of the $\phi_i$.

($\phi$-Forall) By $\phi$-Forall and the IH on $\phi_1$ and $\phi_2$.

($\phi$-SMT-Var) By $\phi$-SMT-Var, using part (2) of the IH on $\Gamma \vdash_{\mathsf{smt}} t$.

($\phi$-SMT-Const) Immediate, by $\phi$-SMT-Const.

($\phi$-SMT-Let) By $\phi$-SMT-Let, using the IH on $v_1$, $v_2$, and $v_3$.

($\phi$-SMT-Ctor) By $\phi$-SMT-Ctor and the IH, using part (2) on $t_j'$ and part (4) on $\phi_i$, observing that the type equivalences are preserved.

($\phi$-SMT-Forall) By $\phi$-SMT-Forall and the IH on $v_1$ and $v_2$.

($\phi$-SMT-UFun) By $\phi$-SMT-UFun and the IH on each of the $v_i$.

**Lemma D.7** (Type well formedness strengthening). *If* $\Gamma, X : \tau, \Gamma' \vdash_m \tau'$ *then* $\Gamma, \Gamma' \vdash \tau'$.

*Proof.* ($t$-Base) Immediate.

($t$-TVar) Immediate: removing the variable binding can't affect $\alpha$.

($t$-ADT) By the IH on each constituent of $D\ \vec{\tau_i}$.

($\tau$-SMT) By the IH on $\Gamma \vdash_{\mathsf{smt}} t$.

($\tau$-Sym) By the IH on $\Gamma \vdash_{\mathsf{smt}} t$. □

**Lemma D.8** (Type variable substitution). *If* $\vdash \Gamma, \alpha, \Gamma'$ *then* $\Gamma \vdash_m \tau'$*:*

1. $\vdash \Gamma, \Gamma'[\tau/\alpha]$*;*
2. *If* $\Gamma, \alpha, \Gamma' \vdash_m \tau$ *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash_m \tau'[\tau/alpha]$*;*
3. *If* $\Gamma, \alpha, \Gamma' \vdash X, \tau' \rhd \Gamma''$ *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash X, \tau'[\tau/\alpha] \rhd \Gamma''[\tau/\alpha]$*; and*
4. *If* $\Gamma, \alpha, \Gamma' \vdash \vec{X_i}, \vec{\tau_i'} \rhd \Gamma''$ *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash \vec{X_i}, \vec{\tau_i'}[\tau/\alpha] \rhd \Gamma''[\tau/\alpha]$.

*Proof.* For parts (1) and (2), by mutual induction on the derivations.

(Γ-Empty) Contradictory: $\cdot \neq \Gamma, \alpha, \Gamma'$.

(Γ-Var) We have $\Gamma' = \Gamma'', X : \tau'$ where $\Gamma, \Gamma'' \vdash_{\text{exp}} \tau'$. By the IH on $\Gamma''$, we know that $\vdash \Gamma, \Gamma''[\tau/\alpha]$; by part (2), we have $\Gamma, \Gamma''[\tau/\alpha] \vdash_{\text{exp}} \tau'[\tau/\alpha]$; and so we have $\vdash \Gamma, \Gamma'[\tau/\alpha]$ by Γ-Var.

(Γ-TVar) We have $\Gamma' = \Gamma'', \beta$; by the IH on $\Gamma''$, we have $\vdash \Gamma, \Gamma''[\tau/\alpha]$; since $\beta[\tau/\alpha] = \beta$, we can apply Γ-TVar to find $\vdash \Gamma, \Gamma'[\tau/\alpha]$ as desired.

($t$-B) Immediate by $t$-B, since $B[\tau/\alpha] = B$.

($t$-TVar) We have $\tau' = \beta$. If $\alpha = \beta$, then we have $\Gamma \vdash_m \alpha[\tau/\alpha]$ by assumption. If $\alpha \neq \beta$, then it must be that $\beta \in \Gamma$ or $\Gamma'$—either way, $\beta$ is unaffected by the substitution and we have $\beta \in \Gamma, \Gamma'[\tau/\alpha]$ and so $\Gamma, \Gamma'[\tau/\alpha] \vdash_m \beta$ by $t$-TVar.

($t$-ADT) By the IH on each premise, followed by $t$-ADT.

($\tau$-SMT) By the IH on $\Gamma, \alpha, \Gamma' \vdash_{\text{smt}} t$ and then by $\tau$-SMT.

($\tau$-Sym) By the IH on $\Gamma, \alpha, \Gamma' \vdash_{\text{smt}} t$ and then by $\tau$-Sym.

For parts (3) and (4), by mutual induction on the derivations.

($X\tau$-Bind) We have $X \notin \text{dom}(\Gamma, \alpha, \Gamma')$, so it must also be the case that $X \notin \text{dom}(\Gamma, \Gamma'[\tau/\alpha])$. We therefore find $\Gamma, \Gamma'[\tau/\alpha] \vdash X, \tau'[\tau/\alpha] \rhd \Gamma, \Gamma'[\tau/\alpha], \tau'[\tau/\alpha]$ by $X\tau$-Bind.

($X\tau$-Check) We have $(\Gamma, \alpha, \Gamma')(X) = \tau'$. Is $X : \tau'$ in $\Gamma$ or $\Gamma'$? Either way we will find $\Gamma, \Gamma'[\tau/\alpha] \vdash X, \tau'[\tau/\alpha] \rhd \Gamma, \Gamma'[\tau/\alpha]$ by $X\tau$-Check.

If $\tau' \in \text{dom}(\Gamma)$, then $\Gamma \vdash \tau'$ and so $\tau'[\tau/\alpha] = \tau'$ $(\Gamma, \Gamma'[\tau/\alpha])(X) = \tau'$ and we have $\Gamma, \Gamma'[\tau/\alpha] \vdash X, \tau' \rhd \Gamma, \Gamma'[\tau/\alpha]$.

If, on the other hand, $\tau' \in \text{dom}(\Gamma')$, then $(\Gamma, \Gamma'[\tau/\alpha])(X) = \tau'[\tau/\alpha]$. We therefore have $\Gamma, \Gamma'[\tau/\alpha] \vdash X, \tau'[\tau/\alpha] \rhd \Gamma, \Gamma'[\tau/\alpha]$.

($\vec{X}\vec{\tau}$-All) By part (3) of the IH on each premise. □

**Lemma D.9** (Regularity; formulas have SMT types). *1. If $\vdash \Gamma$ and $\Gamma(X) = \tau$ then $\Gamma \vdash_{\text{exp}} \tau$.*

*2. If $\vdash \Phi$ then (a) if $f : \forall \vec{\alpha_j}, \vec{\tau_i} \to \tau \in \Phi$ then $\vec{\alpha_j} \vdash_{\text{exp}} \tau$, and (b) if $uf : \vec{t_i'} \to t \in \Phi$ then $\cdot \vdash_{\text{smt}} t$.*

*3. If $\Delta; \Phi; \Gamma \vdash e : \tau$ then $\Gamma \vdash_{\text{exp}} \tau$.*

*4. If $\Delta; \Phi; \Gamma \vdash \phi : \tau$ then $\tau = t$ smt or $\tau = t$ sym and $\Gamma \vdash_{\text{smt}} t$ (and so $\Gamma \vdash_{\text{exp}} \tau$).*

*5. If $\Delta; \Phi; \Gamma \vdash \vec{e_i} : \vec{\tau_i}$ then $\Gamma \vdash_{\text{exp}} \tau_i$.*

*6. If $\Delta; \Phi; \Gamma \vdash \vec{\phi_i} : \vec{\tau_i}$ then $\tau_i = t_i$ smt or $\tau = t_i$ sym and $\Gamma \vdash_{\text{smt}} t_i$ (and so $\Gamma \vdash_{\text{exp}} \tau_i$).*

*Proof.* By induction on the typing derivation.

***Contexts***

(Γ-Empty) Contradictory—there's no way $\cdot$ has a binding for $X$.

(Γ-Var) $\Gamma = \Gamma', Y : \tau$. If $X = Y$, then we know $\Gamma \vdash_{\text{exp}} \tau$ by assumption; otherwise, by the IH on $\Gamma'$.

(Γ-TVar) $\Gamma = \Gamma', \alpha$. By the IH on $\Gamma$.

***Program signatures***

(Φ-Empty) Contradictory—there are no function definitions in $\cdot$.

(Φ-Fun) $\Phi = \Phi', g : \ldots$. For case (a) when $f = g$, then by assumption. Otherwise, by the IH on $\Phi'$.

(Φ-Rel) $\Phi = \Phi', p \subseteq \vec{\tau_i}$. By the IH on $\Phi$.

(Φ-UFun) $\Phi = \Phi', uf' : \vec{t_i'} \to t$. For case (b) when $uf = uf'$, then by assumption. Otherwise, by the IH on $\Phi'$.

***Expressions***

($e$-Var) By the part (1) on $\vdash \Gamma$.

($e$-Const) By assumption, we know that $\Gamma \vdash_{\text{smt}} \text{typeof}(k)$; by Lemma D.5 we can find $\Gamma \vdash_{\text{exp}} \text{typeof}(k)$.

($e$-Let) By the IH on $\Gamma, X : \tau_1 \vdash e_2 : \tau_2$, using strengthening (Lemma D.7) to find that if $\Gamma, X : \tau_1 \vdash_{\text{exp}} \tau_2$ then $\Gamma \vdash_{\text{exp}} \tau_2$.

($e$-Ctor) Since $\Gamma \vdash_{\text{exp}} \tau_j'$, we know by $t$-ADT that $\Gamma \vdash_{\text{exp}} D\ \vec{\tau_j'}$.

($e$-Quote) By the IH on part (4), we know that $\Gamma \vdash_{\text{exp}} \tau$ (and, less relevantly, that $\tau = t$ smt or $t$ sym).

($e$-Rel) Immediate by $t$-B.

($e$-Fun) Since $f : \forall \vec{\alpha_j}, \vec{\tau_i} \to \tau \in \Phi$ and $\vdash \Phi$, we know by part (2) of the IH know that $\vec{\alpha_j} \vdash_{\text{exp}} \tau$. By weakening (Lemma D.6) we can lift that well formedness judgment to $\Gamma$. Since each $\Gamma \vdash_{\text{exp}} \tau'_j$, we can find that $\Gamma \vdash \tau[\tau'_j/\alpha_j]$ by substitution (Lemma D.8).

($e$-If) By the IH on $\Gamma \vdash e_2 : t$.

($e$-Match) By the IH on $\Gamma, \overrightarrow{X_1 : \tau_1[\tau_j/\alpha_j]} \vdash e_1 : \tau$ we have $\Gamma, \overrightarrow{X_1 : \tau_1[\tau_j/\alpha_j]} \vdash \tau$; we can use strengthening (Lemma D.7) to find $\Gamma \vdash_{\text{exp}} \tau$.

***Formulae***

($\phi$-Var) Immediate, with $\tau = t$ sym and $\Gamma \vdash_{\text{smt}} t$ by assumption.

($\phi$-Promote) Since $\Delta; \Phi; \Gamma \vdash \phi : t$ sym, we know that $\Gamma \vdash_{\text{smt}} t$ and so we are correct in yielding $\tau = t$ smt.

($\phi$-Unquote) We have $\Delta; \Phi; \Gamma \vdash e : \tau$; by the IH on part (3), we know that $\Gamma \vdash_{\text{exp}} \tau$; by Lemma D.4 we know that toSMT($\tau$) is a well formed SMT type.

($\phi$-Let) By the IH on $\Gamma \vdash \phi_2 : t_2$ smt.

($\phi$-Ctor) We know that $D\ \vec{t'_j}$ is well formed by $t$-ADT; we can find its translation is well formed by Lemma D.4.

($\phi$-UFun) Since $\vdash \Phi$ and $uf : \vec{t_i} \to t \in \Phi$, we know that $\cdot \vdash_{\text{smt}} t$ by part (2) of the IH and so $\cdot \vdash_{\text{exp}} t$ smt, which we can lift to $\Gamma$ by weakening (Lemma D.6).

($\phi$-Forall) Immediate by $t$-B.

($\phi$-SMT-Var) Immediate, with $\tau = t$ sym and $\Gamma \vdash_{\text{smt}} t$ by assumption.

($\phi$-SMT-Const) Immediate, since we have by assumption that $\cdot \vdash_{\text{smt}}$ typeof($k$).

($\phi$-SMT-Let) By the IH on $\Gamma \vdash v_2 : t_2$ smt.

($\phi$-SMT-Ctor) We know that $D\ \vec{t'_j}$ is well formed by $t$-ADT; we can find its translation is well formed by Lemma D.4.

($\phi$-SMT-Forall) Immediate by $t$-B.

($\phi$-SMT-UFun) Since $\vdash \Phi$ and $uf : \vec{t_i} \to t \in \Phi$, we know that $\cdot \vdash_{\text{smt}} t$ by part (2) of the IH and so $\cdot \vdash_{\text{exp}} t$ smt, which we can lift to $\Gamma$ by weakening (Lemma D.6).

***Vectored expressions and formulas*** By the IH for parts (3) and (4), respectively. □

**Lemma D.10** (SMT value conversion is idempotent)**.** toSMT($v$) = toSMT(toSMT($v$))

*Proof.* By induction on $v$.

($v = k$) We have toSMT($k$) = $c^{\text{SMT}}_{\text{const}}(k)$, which is untouched by a second call to toSMT.

($v = c(\vec{v_i})$) We have toSMT($k$) = $c^{\text{SMT}}_{\text{ctor}}(c, \vec{v_i})$, which is untouched by a second call to toSMT.

($v = c^{\text{SMT}}_{\dots}$) All of these SMT constructors are untouched by toSMT. □

**Lemma D.11** (SMT value conversion is type correct)**.** *If* $\Delta; \Phi; \Gamma \vdash v : \tau$ *then* $\Delta; \Phi; \Gamma \vdash$ toSMT($v$) : toSMT($\tau$).

*Proof.* By induction on the typing derivation. In expression mode, the applicable rules are $e$-Const and $e$-Ctor; only a few typing rules could even have applied to a value in formula mode: the $\phi$-SMT-... rules for the $c^{\text{SMT}}_{\dots}$ constructors and $\phi$-Promote.

($e$-Const) We have $\Gamma \vdash k :$ typeof($k$); since toSMT($k$) = $c^{\text{SMT}}_{\text{const}}(k)$ and toSMT(typeof($k$)) = $k$ smt (since $\Gamma \vdash_{\text{smt}}$ typeof($k$) by assumption), we must show that $\Gamma \vdash c^{\text{SMT}}_{\text{const}}(k) :$ typeof($k$) smt, which we have by $\phi$-SMT-Const.

($e$-Ctor) We have $v = c(\vec{v_i})$ and:

$$\Delta(D) = \forall \vec{\alpha_j}, \{\dots, c : \vec{\tau_i}, \dots\} \qquad \Gamma \vdash_{\text{exp}} \tau'_j \qquad \Gamma \vdash v_i : \tau_i[\tau'_j/\alpha_j]$$

Further, toSMT($c(\vec{v_i})$) = $c^{\text{SMT}}_{\text{ctor}}(c, \overrightarrow{\text{toSMT}(v_i)})$. By the IH on each of these $v_i$, we know that we find appropriate values at appropriately converted types, i.e., $\Gamma \vdash$ toSMT($v_i$) : toSMT($\tau_i[\tau'_j/\alpha_j]$). By Lemma D.4, we know toSMT($\tau_i[\tau'_j/\alpha_j]$) is some well formed SMT type; by Lemma D.2 we know that it is also equivalent to the original type. We are *almost* able to apply $\phi$-SMT-Ctor, but we must pick appropriate $t'_j$. We know that toSMT($\tau'_j$) produces an equivalent (Lemma D.2) and well formed (Lemma D.4) SMT type to $\tau'_j$ of the form $t'_j$ smt or $t'_j$ sym. In either case, let *that* be our $t'_j$. We can now apply $\phi$-SMT-Ctor to find that $\Gamma \vdash$ toSMT($c(\vec{v_i})$) : toSMT($D\ \vec{\tau'_j}$).

($\phi$-Promote) By the IH on $\Gamma \vdash v : t$ sym, we know that $\Gamma \vdash$ toSMT($v$) : toSMT($t$ sym), i.e., $\Gamma \vdash$ toSMT($v$) : $t$ sym. By reapplying $\phi$-Promote we can find that $\Gamma \vdash$ toSMT($v$) : $v$ smt.

($\phi$-SMT-...) Immediate: in each of these cases, toSMT does nothing to the $c^{\mathsf{SMT}}_{\dots}$ constructed value nor to the SMT-type $\tau$ assigned to it (which is $t$ smt in all cases except for $c^{\mathsf{SMT}}_{\mathsf{var}}$).

□

## E Type safety

To prove type safety, we prove two properties for every mode of evaluation: first, it is *safe*, i.e, never yields $\perp$; and second, it is *type preserving*, i.e., well typed inputs yield well typed outputs.

The proofs are fairly conventional. For all but the last step, we prove safety and type preservation simultaneously. We start with expressions and formulas (Lemma E.2), which requires a modest notion of canonical forms (Lemma E.1). Next, we prove that value unification (Lemma E.3) is type preserving, reasoning about unification in general within the lemma showing safety and type preservation for premises (Lemma E.4). After a brief lemma about bindings (Lemma E.5), we can prove that program evaluation is safe (Theorem E.6) and type preserving (Theorem E.7).

**Lemma E.1** (Canonical forms for $t$ sym)**.** *If* $\Delta; \Phi; \Gamma \vdash v : t$ sym *then* $v = c^{\mathsf{SMT}}_{\mathsf{var}}(x, t)$.

*Proof.* The only typing rule that could have applied is *phi*-SMT-Var. □

**Lemma E.2** (Term and formula type safety)**.** *If* $\Delta; \Phi \models \mathcal{W}$ *and* $\Delta; \Phi \vdash \vec{F}$ *and* $\Gamma \models \theta$*, when either:*

1. $\Delta; \Phi; \Gamma \vdash e : \tau$ *and* $\mathcal{W}; \theta \vdash e \Downarrow_e v_\perp$*; or*
2. $\Delta; \Phi; \Gamma \vdash \phi : \tau$ *and* $\mathcal{W}; \theta \vdash \phi \Downarrow_\phi v_\perp$
3. $\Delta; \Phi \vdash \mathsf{fun}\ f(\vec{X_i} : \vec{\tau_i}) : \tau = e$ *and* $\vec{\alpha_j}, \overrightarrow{X_i : \tau_i} \models \theta'$ *and* $\mathcal{W}; \theta' \vdash e \Downarrow_e v_\perp$

*then* $v_\perp = v$ *(i.e.,* $v_\perp \neq \perp$*) and* $\Delta; \Phi; \Gamma \vdash v : \tau$.

*Similarly, when either:*

1. *if* $\Delta; \Phi; \Gamma \vdash e_i : \tau_i$ *and* $\mathcal{W}; \theta \vdash \vec{e_i} \Downarrow_e \vec{v_i}$ *then* $\Delta; \Phi; \Gamma \vdash \vec{v_i}_\perp : \vec{\tau_i}$*; and*
2. *if* $\Delta; \Phi; \Gamma \vdash \phi_i : \tau_i$ *and* $\mathcal{W}; \theta \vdash \vec{\phi_i} \Downarrow_{\vec{\phi}} \vec{v_i}$ *then* $\Delta; \Phi; \Gamma \vdash \vec{v_i}_\perp : \tau$

*then* $\vec{v_i}_\perp = \vec{v_i}$ *(i.e., it is not* $\perp$*) and* $\Delta; \Phi; \Gamma \vdash v_i : \tau_i$.

*Proof.* By mutual induction on derivations and the length of the vectored expressions/formulas, leaving $\theta$ general (for, e.g., $e$-Let and $e$-Match).

***Expressions***

($e$-Var) We have $\Gamma(X) = \tau$; since $\Gamma \models \theta$, we have $\theta(X) = v$ (and so $\Downarrow_e$-Var-E didn't apply). So it must be the case that $\Downarrow_e$-Var applied. We can see further that $\Delta; \Phi; \cdot \vdash v : \tau$, and we are done by weakening (Lemma D.6).

($e$-Const) It must be that $\Downarrow_e$-Const applied, and we immediately see that $v_\perp \neq \perp$ and $k$ is well typed in any well formed context by assumption and $e$-Const.

($e$-Let) We know that $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, X : \tau_1 \vdash e_2 : \tau_2$. By the IH on $e_1$, we know that $\theta; \mathcal{W} \vdash e_1 \Downarrow_e v_1$, so it can't be the case that $\Downarrow_e$-Let-E applied—it must hae been $\Downarrow_e$-Let. By the IH on $e_2$, we know that the final result is also not $\perp$ and is well typed.

($e$-Ctor) We have $\Delta(D) = \forall \vec{\alpha_j}, \{\dots, c : \vec{\tau_i}, \dots\}$ and $\Gamma \vdash e_i : \tau_i[\tau'_j/\alpha_j]$. By the IH, we know that each of the $\vec{e_i}$ must have reduced to non-$\perp$ values, and so $\Downarrow_e$-Ctor-E could not have applied. We can therefore see that each $e_i$ reduces to an appropriately typed $v_i$, and our resulting value is well typed by $e$-Ctor.

($e$-Quote) Only $\Downarrow_e$-Quote could have applied. By the IH, we know that $\phi$ reduces to a non-$\perp$ value $v$ well typed at $\tau$.

($e$-Rel) We know that $p \subseteq \vec{\tau_i} \in \Phi$ and $\Gamma \vdash e_i : \tau_i$. The IH on $\vec{e_i}$ rules out *StepstoE*-Rel-E1; the typing rule rules out the arity mismatch in $\Downarrow_e$-Rel-E2 and the missing relation in $\Downarrow_e$-Rel-E3. So it must be the case that $\Downarrow_e$-Rel-True or $\Downarrow_e$-Rel-False applied; either way, we yield a bool, which is appropriately typed by $e$-Const.

($e$-Fun) We know that $f : \forall \vec{\alpha_j}, \vec{\tau_i} \to \tau \in \Phi$ and $\Gamma \vdash e_i : \tau_i[\tau'_j/\alpha_j]$. The IH on $\vec{e_i}$ rules out *StepstoE*-Fun-E1; the typing rule rules out the arity mismatch in $\Downarrow_e$-Fun-E2 and the missing function in $\Downarrow_e$-Fun-E3. So it must be the case that $\Downarrow_e$-Fun applied. Since $\Delta; \Phi \vdash F$, we know by the IH on part (3) that the resulting value is non-$\perp$ and well typed at $\tau[\tau'_j/\alpha_j]$.

($e$-If) We have $\Gamma \vdash e_1 : \text{bool}$ and $\Gamma \vdash e_2 : \tau$ and $\Gamma \vdash e_3 : \tau$. By the IH on $e_1$, we know that $e_1$ reduces to true or false (since those are the only values of type bool). So we can rule out $\Downarrow_e$-Ite-E1 and $\Downarrow_e$-Ite-E2—we must have stepped by either $\Downarrow_e$-Ite-T or $\Downarrow_e$-Ite-F. The IH on $e_2$ or $e_3$ (respectively) guarantees we step to a non-$\bot$, well typed value.

($e$-Match) We have $\Gamma \vdash e : D\ \vec{\tau_j}$ and $\Delta(D) = \forall \vec{\alpha_j}, \{\dots, c_i : \vec{\tau_k}, \dots\}$ and $\Gamma, \overrightarrow{X_k : \tau_k[\tau_j/\alpha_j]} \vdash e_i : \tau$. The IH on $e$ guarantees that we get a non-$\bot$ value at type $D\ \vec{\tau_j}$, which rules out the error case $\Downarrow_e$-Match-E1, the non-constructor value of $\Downarrow_e$-Match-E2, the mis-named constructor of $\Downarrow_e$-Match-E3, and the arity error of $\Downarrow_e$-Match-E4. So it must be the case that we applied $\Downarrow_e$-Match; by the IH, the matching pattern reduces to a well typed non-$\bot$ value.

***Formulae***

($\phi$-Var) The term $x{:}t$ could only have stepped by $\Downarrow_\phi$-Var to $v = c^{\text{SMT}}_{\text{var}}(x, t)$; we must show $\Delta; \Phi; \Gamma \vdash c^{\text{SMT}}_{\text{var}}(x, t)$—which we have by $\phi$-SMT-Var.

($\phi$-Promote) We have $\Delta; \Phi; \Gamma \vdash \phi : t\ \text{sym}$; by the IH, we know that $\phi$ steps to a non-$\bot$ value $v$ well typed at $t$ sym; by $\phi$-Promote we can see that $v$ is also well typed at $t$ smt.

($\phi$-Unquote) We have $,e$; since $\Delta; \Phi; \Gamma \vdash e : \tau$, we know by the IH that $e$ reduces to a non-$\bot$ value $v$ that is also well typed at $\tau$. We can therefore rule out $\Downarrow_\phi$-Unquote-E, so we must have stepped by $\Downarrow_\phi$-Unquote.

Since $\Delta; \Phi; \Gamma \vdash v : \tau$, we have $\Delta; \Phi; \Gamma \vdash \text{toSMT}(v) : \text{toSMT}(\tau)$ by Lemma D.11, as desired.

($\phi$-Let) Since the constituent parts of the term $\text{let}_\phi\ \phi_1 = \phi_2\ \text{in}\ \phi_3$ are well typed, we can use the IH to see that each $\phi_i$ reduces a non-$\bot$ value $v_i$ of appropriate type, i.e.:

$$\Delta; \Phi; \Gamma \vdash v_1 : t_1\ \text{sym} \qquad \Delta; \Phi; \Gamma \vdash v_2 : t_1\ \text{smt} \qquad \Delta; \Phi; \Gamma \vdash v_3 : t_2\ \text{smt}$$

We can therefore rule out $\Downarrow_\phi$-Let-E1. By Lemma E.1, we know that $v_1$ is appropriately an SMT variable, ruling out $\Downarrow_\phi$-Let-E2. The resulting value is $c^{\text{SMT}}_{\text{let}}(v_1, v_2, v_3)$, which is well typed by $\phi$-SMT-Let.

($\phi$-Forall) Since the constituent parts of the term $\forall \phi_1.\ \phi_2$ are well typed, we can use the IH to see that each $\phi_i$ reduces a non-$\bot$ value $v_i$ of appropriate type, i.e.:

$$\Delta; \Phi; \Gamma \vdash v_1 : t_1\ \text{sym} \qquad \Delta; \Phi; \Gamma \vdash v_2 : \text{bool smt}$$

We can therefore rule out $\Downarrow_\phi$-Forall-E1. By Lemma E.1, we know that $v_1$ is appropriately an SMT variable, ruling out $\Downarrow_\phi$-Forall-E2. The resulting value is $c^{\text{SMT}}_{\text{forall}}(v_1, v_2)$, which is well typed by $\phi$-SMT-Forall.

($\phi$-Ctor) We have $c(\vec{\phi_i})$, where each $\phi_i$ is well typed at $t'_i$ smt, where each parameter of the constructor $c$ is equivalently typed at $\tau_i$ under a well formed substitution of an SMT type $[t'_j/\alpha_j]$. By the IH on each $\phi_i$, we know that each constructor argument reduces to a non-$\bot$ value $v_i$ well typed at $t'_i$ smt; we can therefore rule out $\Downarrow_\phi$-Ctor-E so we must have stepped by $\Downarrow_\phi$-Ctor to $\text{toSMT}(c(\vec{v_i}))$.

Since $c$ isn't an SMT constructor, we know $\text{toSMT}(c(\vec{v_i})) = c^{\text{SMT}}_{\text{ctor}}(c, \vec{v_i})$, which is well typed by $\phi$-SMT-Ctor: given the well typing of the values $\vec{v_i}$, the other premises correspond one-to-one with those of $\phi$-Ctor.

($\phi$-UFun) We have $uf(\vec{\phi_i})$ where each $\phi_i$ is well typed at $t_i$ smt. By the IH, each such $\phi_i$ reduces to a non-$\bot$ value $v_i$ well typed at $t_i$ smt. So $\Downarrow_\phi$-UFun-E could not have applied. We must have stepped by $\Downarrow_\phi$-UFun to $c^{\text{SMT}}_{\text{uf}}(uf, \vec{v_i})$, which is well typed by $\phi$-SMT-UFun.

($\phi$-SMT-...) For each of the SMT constructor rules ($\phi$-SMT-Var, $\phi$-SMT-Const, $\phi$-SMT-Ctor, $\phi$-SMT-Let, $\phi$-SMT-Forall, $\phi$-SMT-UFun), there is only one rule that could have applied: $\Downarrow_\phi$-SMT-Ctor, wherein we immediately step to the same value which remains well typed.

***Functions*** By part (1) on $\vec{\alpha_j}, \overrightarrow{X_i : \tau_i} \vdash e : \tau$, using weakening (Lemma D.6) to recover typing in $\Gamma$.

***Vectored expressions and formulas*** By induction on the vector length, using parts (1) and (2) in each case. □

**Lemma E.3** (Value unification preservation). *If $\Gamma \models \theta$ when either:*

1. *$\Gamma \vdash \vec{X}, \vec{\tau} \rhd \Gamma'$ and $\Gamma \vdash \vec{v} : \vec{\tau}$ and $\theta \vdash \vec{X} \sim \vec{v} : \theta'$; or*
2. *$\Gamma \vdash X, \tau \rhd \Gamma'$ and $\Gamma \vdash v : \tau$ and $\theta \vdash X \sim v \rhd \theta'$;*

*then $\Gamma' \models \theta'$.*

*Proof.* By induction on the derivation of well typing.

($X\tau$-Bind) Only $uv$-Bind-Var could have applied, so we have $\Gamma \vdash v : \tau$ and $\Gamma \models \theta$ and must show that $\Gamma, X : \tau \models \theta[X \mapsto v]$, which we have immediately.

($X\tau$-Check) Here $X \in \Gamma$, so it must be that $\theta(X)$ is defined. One of three rules could have applied:

($uv$-Eq-Var) We have $\Gamma' = \Gamma$ and $\theta' = \theta$, so $\Gamma' \models \theta'$ by assumption.

($uv$-Ctor) By the IH, we know that $\Gamma' models \theta'$.

($uv$-Constant) As for $uv$-Eq-Var, we have $\Gamma' = \Gamma$ and $\theta' = \theta$, so $\Gamma' \models \theta'$ by assumption.

($X\tau$-All) It must be that $\vec{u}\vec{v}$-All applied; by the IH on each sub-derivation, we can find that $\Gamma_i \models \theta_i$, and so $\Gamma' \models \theta'$ in particular.

□

User code will never *directly* trigger a use of $uv$-Eq-Var directly, because the unification rules won't call value unification with a defined LHS (we'd just use $uu$-BB instead). But a use of $\vec{u}\vec{v}$-All could lead to a variable being unified early on and then used again in the same unification process.

**Lemma E.4** (Premise preservation and safety). *If $\Delta; \Phi; \Gamma \vdash P \rhd \Gamma'$ and $\Delta; \Phi \models \vec{F}$ and $\Delta; \Phi \models \mathcal{W}$ and $\Gamma \models \theta$ then if $\vec{F}; \mathcal{W}; \theta \vdash P \longrightarrow \theta'_\perp$ then:*

1. *$\theta'_\perp = \theta'$ (i.e., it is not $\perp$); and*
2. *$\Gamma' \models \theta'$.*

*Proof.* By induction on the premise typing derivation, followed by cases on the step taken.

($P$-PosAtom) We have:

$$p \subseteq \vec{\tau}_i \in \Phi \qquad \Gamma \vdash \vec{X}_i, \vec{\tau}_i \rhd \Gamma'$$

The only rule that could have applied is PosAtom, i.e., $\vec{v} \in \mathcal{W}(p)$ and $\theta \vdash \vec{X}_i \sim \vec{v}_i : \theta'_\perp$. We must show that $\theta'_\perp = \theta'$ and $\Gamma' \models \theta'$.

Since $\Delta; \Phi \models \mathcal{W}$, we know that $\cdot \vdash \vec{v}_i : \vec{\tau}_i$; by weakening we have $\Gamma \vdash \vec{v}_i : \vec{\tau}_i$ (Lemma D.6).

Syntactically, we know that $\vec{X}_i$ are all variables and that $\vec{v}_i$ are all values. For each one, therefore only two unification rules could possibly apply: $uu$-BB ($X_i$ is bound) and $uu$-FB ($X_i$ is free). In particular, $uu$-FF cannot apply, and so we cannot produce $\perp$, so $\theta'_\perp = \theta' = \theta\vec{\theta}_i$. By Lemma E.3, we know that $\Gamma' \models \theta\theta_i$ for each $i$, and so $\Gamma' \models \theta\vec{\theta}_i$.

($P$-NegAtom) We have:

$$p \subseteq \vec{\tau}_i \in \Phi \qquad \Gamma \vdash \vec{X}_i, \vec{\tau}_i \rhd \Gamma$$

Two rules are possible: NegAtom and NegAtom-E. We must show that the latter cannot apply and that the former preserves typing.

Since $\Gamma \models \theta$, it must be that case that each $\vec{X}_i \in \text{dom}(\theta)$, and so NegAtom-E cannot have applied. It remains to be seen that $\Gamma \models \theta'$—but in NegAtom we have $\theta = \theta'$, and so we are done.

($P$-EqCtor-BF) We have:

$$\Delta(D) = \forall \vec{\alpha}_j, \{\dots, c : \vec{\tau}_i, \dots\}$$
$$\Gamma \vdash Y, \tau[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma \qquad \Gamma \vdash \vec{X}_i, \vec{\tau}_i[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma'$$

The only rule that could have applied is EqCtor, where $\theta \vdash Y\ c(\vec{X}_i) : \theta'_\perp$. We must show that $\theta'_\perp = \theta'$ (i.e., it is not $\perp$) and that $\Gamma' \models \theta'$.

Since $\Gamma \vdash Y, \tau[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma$, it must be the case that $Y \in \text{dom}(\Gamma)$ and so $\theta(Y) = v$ (and so $\cdot \vdash v : \tau[\vec{\tau}'_j/\vec{\alpha}_j]$, which also holds under $\Gamma$ thanks to weakening (Lemma D.6)).

Only two rules could have applied to show $\theta \vdash Y\ c(\vec{X}_i) : \theta'_\perp$: $uu$-BF (when some of $\vec{X}_i$ are unbound) or $uu$-BB (when all of the $\vec{X}_i$ are bound). In either case, $uu$-FF can't have a applied, and so $\theta'_\perp = \theta'$.

One of two rules could have applied: $uv$-Eq-Var or $uv$-Ctor.

In the former case, we applied $uu$-BB, because $\theta(c(\vec{X}_i)) = c(\vec{v}_i)$. We have $\theta' = \theta[X \mapsto c(\vec{v}_i)$ and $\Gamma' \models \theta'$ by substitution on $\Gamma \vdash \vec{X}_i, \vec{\tau}_i[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma'$ (Lemma D.8).

In the latter case, we can find that $\Gamma' \models \theta'$ by Lemma E.3 on the assumption that $\Gamma \vdash \vec{X}_i, \vec{\tau}_i[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma'$, and the fact $\theta(Y) = v$ is well typed in $\Gamma$.

($P$-EqCtor-FB) We have:

$$\Delta(D) = \forall\vec{\alpha}_j, \{\dots, c : \vec{\tau}_i, \dots\}$$
$$\Gamma \vdash Y, \tau[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma'$$
$$\Gamma \vdash \vec{X}_i, \vec{\tau}_i[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma$$

The only rule that could have applied is EqCtor, where $\theta \vdash Y\ c(\vec{X}_i) : \theta'_\perp$. We must show that $\theta'_\perp = \theta'$ (i.e., it is not $\perp$) and that $\Gamma' \models \theta'$.

Since $\Gamma \vdash \vec{X}_i, \vec{\tau}_i[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma$, it must be that case $\vec{X}_i \subseteq \text{dom}(\Gamma)$, and so $\theta(c(\vec{X}_i)) = c(\vec{v}_i)$. We know further that $\cdot \vdash c(\vec{v}_i) : \tau[\vec{\tau}'_j/\vec{\alpha}_j]$, which also holds under $\Gamma$ thanks to weakening (Lemma D.6).

Either *uu*-FB or *uu*-BB applied to show $\theta \vdash Y\ c(\vec{X}_i) : \theta'_\perp$: *uu*-FB (when $Y$ is unbound) or *uu*-BB (when $Y$ is bound). Critically, *uu*-FF can't have applied, and so $\theta'_\perp = \theta'$.

We can find that $\Gamma' \models \theta'$ by Lemma E.3 on $\Gamma \vdash Y, \tau[\vec{\tau}'_j/\vec{\alpha}_j] \rhd \Gamma'$ (along with the well typing of $\theta(c(\vec{X}_i))$).

($P$-EqExpr) We have:

$$e \neq c(\vec{e}') \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash Y, \tau \rhd \Gamma'$$

The two possible rules are EqExpr and EqExpr-E. We must show that the latter could not have applied (and so $\theta'_\perp = \theta'$) and that $\Gamma' \models \theta'$. By Lemma E.2, we know that EqExpr-E cannot apply and that $\mathcal{W}; \theta \vdash e \Downarrow_e v$ (and so $\Gamma \vdash v : \tau$).

Since $v$ is a value, either *uu*-FB or *uu*-BB applied, depending on whether or not $Y$ is bound. Either way, *uu*-FF couldn't have applied, and so $\theta'_\perp = \theta'$.

We can find that $\Gamma' \models \theta'$ by Lemma E.3 on $\Gamma \vdash Y, \tau \rhd \Gamma'$ (along with the well typing of $v$). □

**Lemma E.5** (Identical bindings implies containment). *If* $\Gamma \vdash X, \tau \rhd \Gamma$, *then* $X \in \text{dom}(\Gamma)$.
*Similarly, if* $\Gamma \vdash \vec{X}_i, \vec{\tau}_i \rhd \Gamma$, *then* $\vec{X}_i \subseteq \text{dom}(\Gamma)$.

*Proof.* By induction on the derivation.

($X\tau$-Bind) Contradictory: this rule could not have applied, since $\Gamma \neq \Gamma, X : \tau$.

($X\tau$-Check) We have $X \in \text{dom}(\Gamma)$ by assumption.

($\vec{X}\vec{\tau}$-All) By the IH on each of our premises. □

**Theorem E.6** (Program safety). *If* $\Delta; \Phi \vdash \vec{F}_i\ \vec{H}_j$ *and* $\Delta; \Phi \models \mathcal{W}$ *then for all* $H \in \vec{H}_j$, $\neg(\vec{F}_i; \mathcal{W} \vdash H \longrightarrow \perp)$.

*Proof.* The program prog $= \vec{F}_i\ \vec{H}_j$ must have been well typed according to prog-WF, and so we have $\vdash \Delta$ and $\vdash \Phi$ along with derivations for each $F$ and $H$:

$$\begin{array}{ccccc} \Delta; \Phi \vdash F_0 & \dots & \Delta; \Phi \vdash F_i & \dots & \Delta; \Phi \vdash F_n \\ \Delta; \Phi \vdash H_0 & \dots & \Delta; \Phi \vdash H_j & \dots & \Delta; \Phi \vdash H_m \end{array}$$

Let an $H = p(X_k) :- \vec{P}_\ell \in \vec{H}_j$ be given. We know that $\Delta; \Phi \vdash H$ by $H$-Clause, i.e.:

$$\cdot \vdash P_0 \rhd \Gamma_1 \qquad \dots \qquad \Gamma_\ell \vdash P_\ell \rhd \Gamma_{\ell+1} \qquad \dots \qquad \Gamma_p \vdash P_p \rhd \Gamma'$$
$$p \subseteq \vec{\tau}_k \in \Phi \qquad \Gamma' \vdash \vec{X}_k, \vec{\tau}_k \rhd \Gamma'$$

Let $\mathcal{W}$ be given such that $\Delta; \Phi \models \mathcal{W}$. We must show that it is not the case that $\vec{F}_i; \mathcal{W} \vdash H \longrightarrow \perp$, i.e., Clause-E1 and Clause-E2 cannot apply. We can rule out Clause-E1 by Lemma E.4(1: it is not the case that a typesafe premise steps to $\perp$. To rule out Clause-E2, we need to know that if we can build a final substitution, i.e.:

$$\cdot \vdash P_0 \longrightarrow \theta_1 \quad \dots \quad \theta_\ell \vdash P_\ell \longrightarrow \theta_{\ell+1} \quad \dots \quad \theta_p \vdash P_p \longrightarrow \theta$$

then $\vec{X}_k \in \text{dom}(\theta)$. We know that $\vec{X}_k \subseteq \text{dom}(\Gamma')$ by Lemma E.5 on $\Gamma' \vdash \vec{X}_k, \vec{\tau}_k \rhd \Gamma'$; since $\Gamma_p \vdash P_p \rhd \Gamma'$, we know by Lemma E.4(2) that $\Gamma' \models \theta$. We can therefore conclude that $\forall X \in \text{dom}(\Gamma')$, $X \in \text{dom}(\theta)$, and so $\vec{X}_k \in \text{dom}(\theta)$... and Clause-E2 cannot apply. □

**Theorem E.7** (Program preservation). *If* $\Delta; \Phi \vdash \vec{F}_i\ \vec{H}_j$ *and* $\Delta; \Phi \models \mathcal{W}$ *and* $\vec{F}_i; \mathcal{W} \vdash H \longrightarrow \mathcal{W}'$ *for some* $H \in \vec{H}_j$ *then* $\Delta; \Phi \models \mathcal{W}'$.

*Proof.* The program prog $= \vec{F_i}\ \vec{H_j}$ must have been well typed according to prog-WF, and so we have $\vdash \Delta$ and $\vdash \Phi$ along with derivations for each $F$ and $H$:

$$\begin{array}{ccccc} \Delta;\Phi \vdash F_0 & \ldots & \Delta;\Phi \vdash F_i & \ldots & \Delta;\Phi \vdash F_n \\ \Delta;\Phi \vdash H_0 & \ldots & \Delta;\Phi \vdash H_j & \ldots & \Delta;\Phi \vdash H_m \end{array}$$

Let an $H = p(X_k) :- \vec{P_\ell} \in \vec{H_j}$ be given. We know that $\Delta;\Phi \vdash H$ by $H$-Clause, i.e.:

$$\cdot \vdash P_0 \rhd \Gamma_1 \qquad \ldots \qquad \Gamma_\ell \vdash P_\ell \rhd \Gamma_{\ell+1} \qquad \ldots \qquad \Gamma_p \vdash P_p \rhd \Gamma'$$
$$p \subseteq \vec{\tau_k} \in \Phi \qquad \Gamma' \vdash \vec{X_k}, \vec{\tau_k} \rhd \Gamma'$$

Let $\mathcal{W}$ be given such that $\Delta;\Phi \models \mathcal{W}$. It must have been the case that we stepped by Clause, and so:

$$\cdot \vdash P_0 \longrightarrow \theta_1 \quad \ldots \quad \theta_i \vdash P_i \longrightarrow \theta_{i+1} \quad \ldots \quad \theta_n \vdash P_n \longrightarrow \theta$$
$$\mathcal{W}' = \mathcal{W}[p \mapsto \mathcal{W}(p) \cup \theta(\vec{X_j})]$$

By Lemma E.4(2), we know that $\Gamma_i \models \theta_i$ and $\Gamma' \models \theta$. We have $\vec{X_k} \subseteq \text{dom}(\Gamma')$ by Lemma E.5 on $\Gamma' \vdash \vec{X_k}, \vec{\tau_k} \rhd \Gamma'$, we can conclude that $\vec{X_k} \subseteq \text{dom}(\theta)$ and that $\Delta;\Phi;\cdot \vdash \theta(X_k) : \tau_k$ by Lemma E.3 on $\Gamma' \vdash \vec{X_k}, \vec{\tau_k} \rhd \Gamma'$.

To see that $\Delta;\Phi \models \mathcal{W}$, we need to see that adding $\theta(\vec{X_k})$ to $\mathcal{W}(p)$ is safe. We already knew that $p \subseteq \vec{\tau_k} \in \Phi$ and $p \in \text{dom}(\mathcal{W})$; we have $k = k$ immediately, and we have seen that each $\theta(X_k)$ is well typed at $\tau_k$. □