Microsoft Research

Open-World Logic Programs:
A New Foundation for Formal Specifications

MSR Technical Report MSR-TR-2013-55

Open-World Logic Programs: A New Foundation for Formal Specifications

Ethan K. Jackson, Wolfram Schulte, and Nikolaj Bjørner

Microsoft Research, Redmond, WA
{ejackson, schulte, nbjorner}@microsoft.com

Abstract. Recent advances in decision procedures and constraint solvers can enable a new generation of formal specification languages. In this paper we present a new semantic foundation for formal specifications, called *open-world logic programming*, which integrates with state-of-the-art solvers. Analysis, verification, and synthesis problems on open-world logic programs can be converted to constraints by a quantifier-elimination scheme using symbolic execution. This paper presents the features, semantics, and algorithms of open-world logic programs. We have implemented this approach in the FORMULA specification language, which has been used for production-quality specifications and models.

1 Introduction

Formal specification languages rely on automated theorem provers [1], constraint solvers [2], and model checkers [3, 4] for analyzing specifications. Recent advances in decision procedures and constraint solvers [5] can enable a new generation of formal specification languages offering powerful analyses. To this end, we present a new semantic foundation for formal specifications, called *open-world logic programming*, which integrates with state-of-the-art solvers. Analysis, verification, and synthesis problems on open-world logic programs can be converted to constraints by a quantifier-elimination scheme using symbolic execution. This paper presents the features, semantics, and algorithms of open-world logic programs (OLPs). We have implemented this approach in the FORMULA specification language, which has been used for complex specifications and models [6–8]. The contributions of this paper are:

1. **Formal Semantics.** We describe the static, logical, and execution semantics of OLPs based on the *well-founded semantics* for closed-world logic programs [9]. We generalize execution to *symbolic exection* for programs with *symbolic values*.
2. **Analysis.** We show that analysis problems can be specified using OLPs. They can be solved using a quantifier elimination scheme implemented by symbolic execution. The output of symbolic execution is a formula that can be dispatched to state-of-the-art constraint solvers.
3. **Comparisons.** We provide detailed comparisons with the dominant logic programming paradigms. We examine their methods of integrating decision procedures.

This paper is structured as follows: Section 2 explores closed-world and open-world logic programs; it details previous attempts to integrate LP with decision procedures. Section 3 presents the syntax and static semantics of a core open-world language. Section 4 gives the logical semantics of OLPs and Section 5 formalizes execution and symbolic execution. We conclude in Section 6.

2 Extending LP Paradigms With Decision Procedures

Logic programming (LP) has long been used for formal specifications. It is attractive because it is precise, declarative, executable, and expressive (corresponding to various classes of *fixpoint logic*). Its usefulness for formal specifications is still vigorously researched, e.g. in the context of distributed systems [10], access control logics [11, 12], and declarative databases [13]. However, existing LP paradigms do not integrate well with the powerful *decision procedures* arising from constraint solving. The problems are as follows: *Prolog extended with constraints* requires users to syntactically schedule constraint introduction. Consequently, two specifications that appear equivalent may exhibit very different executions. The *Answer Set Programming* (ASP) paradigm, based on *stable models*, performs searches over *finite groundings* of programs. ASP makes only limited use of decision procedures, and lacks generic support for infinite domains even when decision procedures for these domains are available.

We now introduce closed-world and open-world logic programming, and give a detailed account of existing LP paradigms and their integration with decision procedures. In the interest of space, we shall study a small problem that challenges these paradigms. In the text to follow $\boldsymbol{t}$ is a vector of terms and t_i is the i^{th} term of $\boldsymbol{t}$. We write $t[\boldsymbol{x}\backslash\boldsymbol{t}]$ for the simultaneous replacement of variables x_i with terms t_i in t.

2.1 A Brief Summary Of Closed-World Logic Programs

A *closed-world* logic program contains a fixed set of *rules*. Each rule has the form $R(\boldsymbol{t})$:- *body*. $R(\boldsymbol{t})$ is called the *head* of the rule. R is a relation symbol called a *program relation*. The *body* is a conjunction of constraints possibly containing other program relations. Informally, whenever the body of a rule is satisfied for some substitution of variables $body[\boldsymbol{x}\backslash\boldsymbol{t'}]$, then the relation R must be true for all elements of the form $\boldsymbol{t}[\boldsymbol{x}\backslash\boldsymbol{t'}]$. *Negation* is the ability to test $\neg R(\boldsymbol{t})$ in the body of a rule. A *fact* is a rule whose body is empty; such a body is treated as true. The exact semantics of programs vary between paradigms. We shall discuss this in more detail in the following sections.

Program execution determines members of a relation R by applying rules. In *forward chaining* the rules are repeatedly applied, starting from facts, building up the elements of R. Execution terminates when no rule extends any relation further, i.e. a least fixpoint is reached. *Backwards chaining* tries to prove that some element of the form $\boldsymbol{t}$ is in R by working backwards through the rules, from heads to bodies until reaching facts. A *query operation* on a closed-world

program decides whether a relation contains some element. Query operations can be evaluated by either forward or backwards chaining.

2.2 The Challenge Problem

In order to place OLP in context of various LP paradigms, we will specify and decide the following property:

Is every list of four integers sortable by adjacent compare-and-swap operations?

More precisely, let l be a quadruple of integers $\langle x_1, x_2, x_3, x_4 \rangle$ and the i^{th} compare-and-swap operation $\curvearrowright^i$ be:

$$\curvearrowright^i (\langle x_1, \ldots x_i, x_{i+1} \ldots, x_n \rangle) \stackrel{def}{=} \begin{cases} \langle x_1, \ldots x_{i+1}, x_i \ldots, x_n \rangle, & \text{if } x_{i+1} < x_i, \\ \langle x_1, \ldots x_i, x_{i+1} \ldots, x_n \rangle, & \text{otherwise.} \end{cases}$$

Does the following hold?

$$\forall l \in \mathbb{Z}^4.\ \exists \sigma_1, \ldots, \sigma_k \in \{\curvearrowright^1, \curvearrowright^2, \curvearrowright^3\}.\ sorted(\sigma_k(\ldots \sigma_1(l) \ldots)).$$

The answer is clearly 'yes', but our goal is to specify and prove it with a logic program. If the property were false the LP engine should construct a counterexample, i.e. a quadruple which can never be sorted using these operations. Though deceptively simple, this problem has three sources of difficulty: (1) The integers are unbounded; a decision procedure for linear inequalities is required. (2) The number k of compare-and-swaps is unknown; the reachable configurations must be stated recursively[1]. (3) The reachability relation must be negated to check for a counterexample. In other words, *constraints*, *recursion*, and *negation* are three sources of expressiveness and challenges in logic programs.

2.3 Open-World Logic Programming

Open-world logic programming removes the closed-world assumption. It allows some parts of programs to be unspecified or *open*. This generalization enables OLPs to succinctly specify complex analysis problems while utilizing decision procedures to solve them. Here are the key attributes of OLPs:

- **Negation.** The use of negation is restricted. Consequently, OLPs can be formalized in a straightforward manner.
- **Closure.** A *closure* of a program is an extension of it by ground facts. After a program is closed, nothing is assumed to be open. Closed programs are well-behaved: they have unique and finite least fixpoints.
- **Query.** A query operation on an OLP searches for a closure where the goal is satisfied.

[1] There is an obvious upper-bound on k, but we expected the underlying proof machinery to happen upon this fact.

```
input     ::= new (Integer, Integer, Integer, Integer).
trace     ::=     (Integer, Integer, Integer, Integer).
cntrexmp  ::=     (Integer, Integer, Integer, Integer).

cntrexmp(W, X, Y, Z) :- input(W, X, Y, Z),
                        no { A, B, C, D | trace(A, B, C, D), A <= B, B <= C, C <= D }.
trace(W, X, Y, Z)    :- input(W, X, Y, Z).
trace(X, W, Y, Z)    :- trace(W, X, Y, Z), W >X.
trace(W, Y, X, Z)    :- trace(W, X, Y, Z), X >Y.
trace(W, X, Z, Y)    :- trace(W, X, Y, Z), Y >Z.
```

Fig. 1. FORMULA OLP: Deciding a sorting property.

We demonstrate these ideas by solving the sorting problem with an OLP. Figure 1 shows the open-world specification of the sorting problem written in our FORMULA language. The first three lines of the program are type declarations for *input*, *trace*, and *cntrexmp*. The keyword 'new' marks *input* as an open type. A closure of this program can contain some *input* facts, but not *trace* or *cntrexmp* facts. The *trace* rules compute the traces generated by sorting the *input* fact of a closure. For instance, if $input(v_1, v_2, v_3, v_4)$ is in the closure, then so is $trace(v_1, v_2, v_3, v_4)$. The following three trace rules compute new traces by applying compare-and-swap operations to the current traces. These are called *recursive* rules, because they compute *trace* by examining *trace*. The *cntrexmp* rule is triggered if there is an input list that never generates a sorted trace. This rule uses negation to check that no sorted traces exists. FORMULA negations appear as set expressions. The negation is true if the set expression evaluates to the empty set.

A closure is called a *model* in FORMULA. Each model has: (1) a name, (2) a reference to the OLP it closes, (3) and a list of facts that close the program. For example:

```
model M of Sort { input(3, 2, 6, 5). }
```

Given a closure, a unique least fixpoint can be computed by applying rules. The least fixpoint of model M contains the following:

$$\left\{ \begin{array}{c} input(3,2,6,5), trace(3,2,6,5), \\ trace(2,3,6,5), trace(3,2,5,6), trace(2,3,5,6) \end{array} \right\}. \tag{1}$$

Open-world queries. An open-world query searches for a closure where a goal is true. For instance, the sorting problem can be decided by the following query: "Find a closure containing exactly one *input* term whose least fixpoint contains a *cntrexmp* term." The algorithms presented here solve such a query as follows: First, the program is closed with *symbolic facts*, e.g.:

$$\mathsf{input}(\star_1, \star_2, \star_3, \star_4).$$

Each $\star_i$ is a *symbolic value*; its exact value is unknown at the time of execution. Symbolic values can be substituted by values from infinite domains, e.g. the set of all integers. Next, program execution is generalized to handle symbolic values. *Symbolic execution* accumulates quantifier-free constraints, which encode how choices for symbolic values influence the least fixpoint. Finally, the quantifier-free constraint for the goal is solved using a collection of decision procedures. FORMULA uses the *Z3* SMT solver [14], which soundly combines a variety of decision procedures. If a solution is found, then a concrete closure is constructed by substituting symbolic values, otherwise the goal is unsatisfiable. The time to solve the sorting problem is dominated by symbolic execution, which takes a fraction of a second. Our technique does not require bounding the range of integers, nor does it rely on users writing the rules in a particular fashion. These desirable properties do not hold for traditional approaches.

2.4 The Classical Approach: Prolog With Constraints

The most classical of LP paradigms is *Prolog*, which implements a backwards chaining engine. *SWI Prolog* is an implementation of Prolog that can be extended with constraint solvers [15], so it is suitable for specifying and proving the sorting property. SWI Prolog illustrates a standard proof strategy: (1) Represent a trace of the sorting algorithm as a sequence of lists, (2) use rules to specify all such traces, (3) and perform a backwards search for a counterexample. The difficulty will be that backwards chaining must consider an infinite number of traces. Figure 2 shows the specification in SWI Prolog. The syntax $[e_1, \ldots, e_n|T]$ is a sequence whose first n elements are $e_1, \ldots, e_n$ and whose remaining elements are the (possibly empty) sub-sequence T.

Recursion. As before, the last four rules compute a *trace* relation, where each element of *trace* is a sequence of swaps applied to some initial list. The first trace rule admits every possible list as an initial trace, and the following three trace rules describe how to apply the compare-and-swap operations to a sub-trace. Table 1 shows a few elements of the trace relation starting from the list $\langle 3, 2, 6, 5\rangle$.

Negation. As before, negation permits a rule to test absence of some elements from a program relation. In SWI Prolog, negation is indicated by the operator '\+'. The first rule uses negation to find counterexamples. Some list $l \stackrel{def}{=} \langle w, x, y, z\rangle$ ($somelist(W, X, Y, Z)$) is a counterexample if it is *not* eventually sorted. A list l is eventually sorted ($evntsorted(W, X, Y, Z)$) if there is a trace starting from l that contains a sorted list. Formalizing negation in LP is non-trivial because of recursion. Informally, this negation tests for membership in the least fixpoint of *trace*, which contains all possible sorting traces.

Constraints. To prove the sorting property, we ask Prolog to find an element in the *cntrexmp* relation via a query with the goal $cntrexmp(W, X, Y, Z)$. If a counterexample can be found, then the variables will be replaced with values giving a member of *cntrexmp*. If *cntrexmp* is empty then we would like the query operation to terminate with the result *false*. Prolog starts from the goal

```
cntrexmp(W,X,Y,Z) :- somelist(W,X,Y,Z), \+evntsorted(W,X,Y,Z).
evntsorted(W,X,Y,Z):- sorted(A,B,C,D), trace([[A,B,C,D] | _ ], [W,X,Y,Z]).
sorted(W, X, Y, Z)  :- {W =< X}, {X =< Y}, {Y =< Z}.
somelist(W, X, Y, Z).

trace([[W,X,Y,Z]], [A,B,C,D])                :- {W = A}, {X = B}, {Y = C}, {Z = D}.
trace([[X,W,Y,Z], [W,X,Y,Z] |T], [A,B,C,D]):- {W >X}, trace([[W,X,Y,Z] |T], [A,B,C,D]).
trace([[W,Y,X,Z], [W,X,Y,Z] |T], [A,B,C,D]):- {X >Y},  trace([[W,X,Y,Z] |T], [A,B,C,D]).
trace([[W,X,Z,Y], [W,X,Y,Z] |T], [A,B,C,D]):- {Y >Z},  trace([[W,X,Y,Z] |T], [A,B,C,D]).
```

Fig. 2. SWI Prolog: Deciding a sorting property.

Swap Applications	LP Representation
initial list $l \stackrel{def}{=} \langle 3,2,6,5\rangle$	$trace([[3,2,6,5]],$ $[3,2,6,5])$
$\curvearrowright^1 (l)$	$trace([[2,3,6,5],[3,2,6,5]],$ $[3,2,6,5])$
$\curvearrowright^3 (l)$	$trace([[3,2,5,6],[3,2,6,5]],$ $[3,2,6,5])$
$\curvearrowright^3 (\curvearrowright^1 (l))$	$trace([[2,3,5,6],[2,3,6,5],[3,2,6,5]],$ $[3,2,6,5])$
$\curvearrowright^1 (\curvearrowright^3 (l))$	$trace([[2,3,5,6],[3,2,5,6],[3,2,6,5]],$ $[3,2,6,5])$

Table 1. Members of *trace* generated from the list $\langle 3,2,6,5\rangle$.

and works backwards to find a member of *cntrexmp*. In doing so, it encounters the negated membership test of *evntsorted* for every possible list. Standard backwards chaining cannot terminate for this query because it must consider an infinite number of traces.

Non-termination can be overcome by extending Prolog with other kinds of constraints and using decision procedures to prune many subgoals at once. In this example, the constraints are solved by a decision procedure for linear inequalities over the rationals [16]. Users write these extra constraints in curly braces, e.g. $\{W > X\}$. They are accumulated in left-to-right order as Prolog attempts to prove the subgoals of a body. If the current set of constraints becomes unsatisfiable, then backwards chaining can terminate the current subgoal and backtrack. In our example all branches of the proof tree become unsatisfiable after a few steps, and so the query operation can terminate without explicitly considering every concrete trace.

Prolog as a specification language. This approach is fast and general, but it comes at a cost: Prolog fixpoints are often infinite, so the user must guide proof search carefully. This is accomplished by the syntactic placement of constraints within rules, and rules within programs. This placement schedules the introduction of constraints and satisfiability checks. For example, if any of the trace rules had their constraints placed later in the body, then the query operation would no longer terminate:

```
trace([[X,W,Y,Z], [W,X,Y,Z] |T], [A,B,C,D]) :-
                    trace([[W,X,Y,Z] |T], [A,B,C,D]), {W >X}.
```

Backwards chaining would forever expand the *trace* subgoal before accumulating the additional constraint $\{W > X\}$.

In summary, classical Prolog extended with constraints utilizes many solvers and incorporates many extensions, such as *constraint handling rules* [17], to dispatch constraints. However, it relies on the user to carefully construct the search strategy within the operational semantics of Prolog. Similar programs can be constructed in the *Ciao* [18] system and the *ECLiPSe* language [19]. These features make Prolog less attractive for formal specifications, because small syntactic perturbations can dramatically affect execution / analysis [20]. In addition, the closed-world assumption of Prolog requires traces to be specified in a more complicated manner, i.e. sequences of lists.

2.5 Answer Set Programming

Answer set programming (ASP) is used for specifying and solving constraint satisfaction problems. ASP takes a different approach from standard Prolog by: (1) unifying the logical and execution semantics of programs, (2) giving semantics to programs with arbitrary negation, (3) and alleviating users from manually encoding search strategies. It accomplishes these goals by observing that programs with arbitrary negation exhibit a set of minimal fixpoints, called *stable models* [21]. The primary job of an ASP engine is to search for stable models. If search problems can be rephrased through stable models, then users do not need to implement search strategies in their programs [20].

Consider the ASP program in Figure 3 written for the *CLASP* ASP system [22]. It also contains four rules specifying traces, but these rules appear more like the OLP rules than Prolog rules. Also unlike Prolog, the program has been written so the *trace* relation contains the traces of only *one* input list, as opposed to all inputs lists. Every stable model shall correspond to a different choice of input list. The last rule in the program does not have a head. It is called a *consistency rule* and its body should never be true. If a stable model exists for this program, then this model is a counterexample, i.e. the chosen input list is never sorted. The ASP engine should either return a stable model for this program, or state that the program is *unsatisfiable*. As with OLP, the syntactic presentation of the rules does not impact the search process.

Generalized negation. The requirement that a stable model contains exactly one input is stated using a *cardinality constraint*:

```
1 { input(W,X,Y,Z) : num(W), num(X), num(Y), num(Z) } 1.
```

It requires the *input* relation to have at least and at most one element. The domain of *input* is restricted to a finite range of integers via the *num* relation. The upper bound on the cardinality constraint corresponds to the following rule employing generalized negation:

```
input(W,X,Y,Z) :- num(W), num(X), num(Y), num(Z), not input(_,_,_,_).
```

Notice that the rule recursively depends on *input* through a negation. Conceptually, once $input(W, X, Y, Z)$ is chosen, it forces the *input* relation to be false

```
num(1..100).
1 { input(W,X,Y,Z) : num(W), num(X), num(Y), num(Z) } 1.

trace(W, X, Y, Z) :- input(W, X, Y, Z).
trace(X, W, Y, Z) :- trace(W, X, Y, Z), W >X.
trace(W, Y, X, Z) :- trace(W, X, Y, Z), X >Y.
trace(W, X, Z, Y) :- trace(W, X, Y, Z), Y >Z.
                  :- trace(W, X, Y, Z), W <= X, X <= Y, Y <= Z.
```

Fig. 3. CLASP ASP: Deciding a sorting property.

for all other places. Therefore, this program gives rise to a set of stable models, each of which chooses a single input and then contains all the traces consistent with this choice.

Model Search. Stable models are defined for *ground programs*, which are programs without variables. Searching for stable models on ground programs is NP-complete and engines employ SAT-like algorithms [23]. Most real-world programs are *non-ground* (e.g. Figure 3), but are grounded as a pre-processing step. *Grounding* expands each rule into many variable-free rules for all possible substitutions of variables by values. For example, the second trace rule is grounded as:

```
trace(1, 1, 1, 1) :- trace(1, 1, 1, 1), 1 > 1.
trace(1, 2, 1, 1) :- trace(2, 1, 1, 1), 2 > 1.
                  ⋮
```

Constraints can now be simplified by evaluation. The first expansion is discarded and the second is simplified to `trace(1, 2, 1, 1) :- trace(2, 1, 1, 1).` Grounding is clearly expensive. Our program restricts integers to the interval $[1, 100]$, but generates 100 million expansions per trace rule during the grounding phase. It quickly runs out of resources when using the state-of-the-art *GrinGo* grounder [24].

ASP engines as constraint solvers. As a specification language, ASP also has clear advantages over Prolog. However, grounding does not make use of modern decision procedures. Combining ASP with other decision procedures is a challenge, because the stable model semantics must be preserved. There have been attempts to *lazily ground* programs by combining grounding with finite domain constraint solvers. The approach is to mark those program variables that should be handled by an external solver [25, 26]. Only some decision procedures are compatible in order to preserve the stable model semantics.

We tried a slightly modified version of the same program using the *Clingcon* ASP engine, which employs the *Gecode* finite domain solver [27]. It is more scalable, but still exhibits exponential slowdown as the integer range is increased. We observed around 1 minute to find the first stable model for the integer range $[-10^5, 10^5]$ and 16 minutes for $[-10^6, 10^6]$, which are small fractions of the full

Programs	Π	$::= E^* \mid R^*$
Type equations	E	$::= \alpha \approx \tau$
Type terms	τ, τ'	$::= \alpha \mid \beta \mid c \mid \tau \cup \tau' \mid f(\tau^+)$
Base types	β	$::= \mathit{Real} \mid \mathit{Integer} \mid \mathit{Natural} \mid \ldots \mid \mathit{String}$
Rules	R	$::= f(t^+) \leftarrow p^* \mid c \leftarrow p^*,$ for c not a built-in constant
Predicates	p, q	$::= K(x) \mid r(t, t') \mid no\{q^*\}$
Terms	t, t'	$::= c \mid x \mid f(t^+) \mid o(t^+)$
Operators	o	$::= + \mid - \mid * \mid \ldots \mid \pi_1 \mid \pi_2 \mid \ldots \mid \mathit{count} \mid \mathit{sum} \mid \ldots$
Relations	r	$::= = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid <: \mid \ldots$
Constants	c	$::= -1 \mid 0.5 \mid \frac{2}{3} \mid \ldots \mid$ `"foo"` $\mid \ldots \mid \mathit{true} \mid \mathit{false} \mid \ldots$

Data constructors $f, g, \ldots$, Variables $x, y, \ldots$, Data types $\alpha, \alpha_f, \ldots$

Fig. 4. Grammar of core programs.

32-bit integer range. In conclusion, ASP leads to cleaner specifications and applies sophisticated techniques to find stable models. However, the stable model semantics makes it difficult to avoid grounding and to incorporate other decision procedures. The OLP approach avoids the stable model semantics in favor of defining open-world programs.

3 Core Programs and Static Semantics

In this section we present the syntax and static semantics of a core open-world logic programming language. FORMULA programs may use a more convenient syntax, but they are translated back into core form. Figure 4 shows the syntax of our core language.

Data types. Data types are important for OLP, because they ensure closures contain meaningful facts. Programs can define new data types using type equations. A type equation E relates a *data type symbol* α with a set of values. Base types such as *Real* and *String* are predefined and name the sets of all real and string values. A *data constructor* is a function for building complex data. Users may introduce new data constructors and write type equations over them. For instance, the type declaration for *input* in Figure 1 introduces the *input* data constructor and a type equation:

$$\alpha_{\mathsf{input}} \approx \mathit{input}(\mathit{Integer}, \mathit{Integer}, \mathit{Integer}, \mathit{Integer}),$$

The α_{input} type stands for the set of all *input* terms with integer arguments (as opposed to *input* terms with string arguments). Because data constructors are first-class, users do not need separately define program relations. We assume an implicit unary program relation K, called the *knowledge set*, containing complex data. If K were made explicit, then FORMULA rules would appear as:

```
K(trace(W, X,Y, Z)) :- K(input(W, X, Y, Z)).
```

Also, the least fixpoint reported in Equation 1 are all elements of K.

A rule may constrain variables to range over types using the *subtyping* relation ('<:'). For example, $x <: \alpha_{\mathsf{input}}$ constrains x to be an *input* applied to four integers. The Prolog / ASP examples do not have explicit data types [28]. However, the use of arithmetic relations forces variables to be instantiated with numeric types. For a full formalization of these data types see [29].

Rules. A rule is a conjunction of predicates. A predicate $K(x)$ is satisfied if x is a member of the knowledge relation K. A predicate $r(t, t')$ is satisfied according to the underlying theory for r. For example, r might be the arithmetic relation '$<$' or '$\geq$'. It might be the equality '$=$' or disequality '$\neq$' relation. Importantly, r is not a program relation; its meaning is fixed and users cannot redefine it. Similarly, operators such as '$+$' or 'π_1' have fixed meanings. (A *selector* π_i extracts the i^{th} argument of a term, e.g. $\pi_1(input(1, 2, 3, 4)) = 1$.) The exact set of relations and operators depends on the decision procedures available. We view these as parameters of the language.

Negation. The last built-in predicate is the *no* predicate. It takes a list of predicates surrounded by a lexical scope. A predicate $no\{\boldsymbol{q}\}$ is true if there is no substitution satisfying the predicates $\boldsymbol{q}$ for those variables *introduced* in the scope. A variable x is introduced in the scope $\{\boldsymbol{p}\}$ if x appears in some p_i and p_i is not of the form $no\{\boldsymbol{q}\}$. For example, this FORMULA rule:

```
cntrexmp(W, X, Y, Z) :- input(W, X, Y, Z),
        no { A, B, C, D | trace(A, B, C, D), A <= B, B <= C, C <= D }.
```

It is translated into core form as follows:

$$cntrexmp(\pi_1(l), \pi_2(l), \pi_3(l), \pi_4(l)) \leftarrow$$
$$K(l), l <: \alpha_{\mathsf{input}}, no\{\ K(t), t <: \alpha_{\mathsf{trace}}, \pi_1(t) \leq \pi_2(t), \pi_2(t) \leq \pi_3(t), \pi_3(t) \leq \pi_4(t)\}.$$

3.1 Static semantics

Not all syntactically correct programs can be handled by our framework. We impose additional *safety* and *stratification* conditions on programs.

Safety. Intuitively, a rule is safe if it does not force an infinite number of elements into K and it does not examine an infinite number of elements. Practically, safety is phrased as a syntactic condition on rules.

Definition 1 (Safe Rule)**.** A rule is safe if whenever a variable x is introduced in a scope, then that scope contains the predicate $K(x)$. Also, every variable appearing in a head must be introduced at the top-most scope of the rule.

The previous translation of the *cntrexmp* rule is an example of safety. The variables w, x, y, z in the head were replaced with expressions over l and the predicate $K(l)$ appears in the body. The FORMULA compiler automatically translates programs into safe rules, or reports an error if a safe orientation of the variables

cannot be found. Prolog has weaker safety conditions because it allows infinite fixpoints. For example, Prolog would allow the fact:

$$input(w, x, y, z) \leftarrow .$$

where all possible input terms are placed into K. ASP has stronger restrictions on safety. Notice that many variables in the ASP example were explicitly guarded by the *num* relation.

Stratification. Stratification restricts negation to guarantee a unique least fixpoint. The general idea is to track dependencies between rules. A stratified program disallows cyclic dependencies through negations. Suppose R and R' are rules, then let *relabel* be a one-to-one function from variables to variables. Assume no variable from R is relabeled into a variable in R'. We lift *relabel* onto rule syntax so that $relabel(s)$ produces identical syntax to s, except that all variables are relabeled. The rules R, R' may depend on each other positively, negatively, or not at all:

Definition 2 (Positive dependency)**.** Rule R' positively depends on R, written $R \rightarrow^{+} R'$ if: (1) R is the rule $t \leftarrow \boldsymbol{p}$ and R' is the rule $t' \leftarrow \boldsymbol{q}$. (2) Some q_i is the predicate $K(x)$. (3) The composition is weakly consistent[2]: $relabel(\boldsymbol{p}), \boldsymbol{q}, x = relabel(t)$.

Definition 3 (Negative dependency)**.** Rule R' negatively depends on R, written $R \rightarrow^{-} R'$ if: (1) R is the rule $t \leftarrow \boldsymbol{p}$ and R' is the rule $t' \leftarrow B_1\ no\{\ \boldsymbol{q}\ \}\ B_2$. (2) Some q_i is the predicate $K(x)$. (3) The composition is weakly consistent: $relabel(\boldsymbol{p}), B_1\ no\{\boldsymbol{q}, x = relabel(t)\}\ B_2$.

The partial bodies B_1 and B_2 split the body of R' to expose an arbitrarily deep negation. A dependency is detected if rule R might produce an element in K that is used to trigger another rule R'. The accuracy of the dependency analysis is tunable; various LP languages implement different versions of this analysis.

Definition 4 (Stratified Programs)**.** An OLP program is stratified if there exists an order $\mathcal{O}$ on rules satisfying:

$$R \rightarrow^{+} R' \Rightarrow \mathcal{O}(R) \leq \mathcal{O}(R') \quad \text{and} \quad R \rightarrow^{-} R' \Rightarrow \mathcal{O}(R) < \mathcal{O}(R'). \tag{2}$$

Such an order exists iff there is no cyclic dependency through negation. This order will be used to formalize programs, and to translate open-world queries into quantifier-free constraints. ASP does not require the strict inequality $\mathcal{O}(R) < \mathcal{O}(R')$ for negative dependencies.

4 Logical Semantics

One advantage of using logic programming for formal specifications is that programs can be directly translated into logic. In this section we give the logical

[2] By *weakly consistent*, we mean satisfiable according to some over-approximation of satisfiability.

$$\mathcal{L}[\![\Pi_{clo}]\!] \stackrel{def}{=} \left(\bigwedge_{E\in\Pi_{clo}} \mathcal{T}[\![E]\!]\right) \wedge \left(\bigwedge_{R\in\Pi_{clo}} \mathcal{R}[\![R, intro(R), \mathcal{O}(R)]\!]\right)$$

$$\begin{array}{ll}
\mathcal{T}[\![\alpha \approx \tau]\!] \stackrel{def}{=} \mathcal{T}[\![\alpha]\!] = \mathcal{T}[\![\tau]\!] & \mathcal{T}[\![c]\!] \stackrel{def}{=} \{c\} \\
\mathcal{T}[\![\tau \cup \tau']\!] \stackrel{def}{=} \mathcal{T}[\![\tau]\!] \cup \mathcal{T}[\![\tau']\!] & \mathcal{T}[\![\beta]\!] \stackrel{def}{=} X, \text{ where } X \text{ is a set such as } \mathbb{Z} \\
\mathcal{T}[\![f(\boldsymbol{\tau})]\!] \stackrel{def}{=} \{f(\boldsymbol{t}) \mid t_i \in \mathcal{T}[\![\tau_i]\!]\} & \mathcal{T}[\![\alpha]\!] \stackrel{def}{=} \alpha
\end{array}$$

$$\mathcal{R}[\![t \leftarrow \boldsymbol{p}, \boldsymbol{x}, n]\!] \stackrel{def}{=} \forall \boldsymbol{x}. \left(\bigwedge_{p_i} \mathcal{R}[\![p_i, \boldsymbol{x}, n, +]\!] \Rightarrow K(t, n)\right)$$

$$\mathcal{R}[\![no\{\boldsymbol{q}\}, \boldsymbol{x}, n, \pm]\!] \stackrel{def}{=} \forall \boldsymbol{y}. \neg(\bigwedge_{q_i} \mathcal{R}[\![q_i, \boldsymbol{z}, n, -]\!]), \qquad \begin{cases} \boldsymbol{y} \stackrel{def}{=} intro(\boldsymbol{q}) - \boldsymbol{x}, \\ \boldsymbol{z} \stackrel{def}{=} intro(\boldsymbol{q}) \cup \boldsymbol{x} \end{cases}$$

$$\begin{array}{l}
\mathcal{R}[\![K(y), \boldsymbol{x}, n, +]\!] \stackrel{def}{=} \exists i.\ (i \leq n \wedge K(y, i)) \\
\mathcal{R}[\![K(y), \boldsymbol{x}, n, -]\!] \stackrel{def}{=} \exists i.\ (i < n \wedge K(y, i)) \\
\mathcal{R}[\![r(t, t'), \boldsymbol{x}, n, \pm]\!] \stackrel{def}{=} r(t, t')
\end{array}$$

Fig. 5. Translation of closures into logic.

Open-World Query	Example Result
$query(\Pi_{Sort}, trace(x, x, x, x))$	$(\{input(1,1,1,1)\}, trace(1,1,1,1))$
$query(\Pi_{Sort}, trace(1, 2, x, x))$	$(\{input(2,4,1,4)\}, trace(1,2,4,4))$
$query(\Pi_{Sort}, trace(x, y, x, y))$	$(\{input(3,3,2,2)\}, trace(3,2,3,2))$
$query(\Pi_{Sort}, cntrexmp(w, x, y, z))$	unsatisfiable

Table 2. Example results for several open-world queries on the sorting program.

semantics of OLPs using the ideas of the *well-founded semantics* [9]. The formalization works as follows: Let $\mathcal{L}[\![]\!]$ be a translator from closures to logic. A structure I is one possible interpretation of a closure Π_{clo} if it satisfies this translation in the usual sense, i.e. $I \models \mathcal{L}[\![\Pi_{clo}]\!]$. However, the intended meaning of Π_{clo} is the *least* such interpretation $lm(\Pi_{clo})$:

$$lm(\Pi_{clo}) \stackrel{def}{=} min\{\ I \mid I \models \mathcal{L}[\![\Pi_{clo}]\!]\ \}. \tag{3}$$

We shall define *min* shortly. A query operation on an OLP Π with goal g returns the set of all extension/term pairs (F, t) whereby closing Π with facts F yields a least interpretation containing the goal.

$$query(\Pi, g) \stackrel{def}{=} \{\ (F, t) \mid lm(\Pi \cup F) \models K(t) \text{ and } t = g[\boldsymbol{x}\backslash\boldsymbol{t}]\ \}. \tag{4}$$

Table 2 shows results for several open-world queries on the sorting program.

Figure 5 shows the translator $\mathcal{L}[\![]\!]$. It translates a program into a conjunction of formulas. The first part translates type equations ($\mathcal{T}[\![]\!]$) and the second part

Init:	$(\emptyset, min(\mathcal{O}))$	
Step:	$(K, n) \Longrightarrow (K \cup \{(t[\boldsymbol{x} \backslash \boldsymbol{t}], n)\}, n)$	If $\begin{cases} \mathcal{O}(R) = n,\ \boldsymbol{x} \stackrel{def}{=} intro(R), \\ R \text{ is } t \leftarrow \boldsymbol{p}, \text{and} \\ K \models (\boldsymbol{x} = \boldsymbol{t}) \wedge \bigwedge_{p_i} \mathcal{R}[\![p_i, \boldsymbol{x}, n]\!] \end{cases}$
Fix:	$(K, n) \Longrightarrow (K, n+1)$	If no step extends K.

Fig. 6. Concrete Execution

translates rules ($\mathcal{R}[\![]\!]$). An interpretation I assigns a set of values α^I to each type symbol α. Rules are slightly more complicated to translate. To correctly translate a rule sub-expression the translator must know: (1) the variables $\boldsymbol{x}$ introduced in earlier scopes, (2) the stratification number of the rule, (3) and whether a predicate appears under a negation (denoted '$-$'). The knowledge relation is translated into a binary relation $K(t, n)$ that holds if t is placed into K by a rule at stratum n. A negation can only examine K at a stratum lower than the rule in which it appears. This particular translation guarantees that all programs have a unique least interpretation:

Definition 5 (Interpretation Order)**.** Let I and I' be two interpretations of Π_{clo}. Then $I \leq I'$ if one of the following holds:

1. I is the same interpretation as I'.
2. There is some α where $\alpha^I \subset \alpha^{I'}$.
3. Both interpretations are identical for type symbols, but for the first stratum n where I and I' differ: Whenever $K^I(t, n)$ holds then $K^{I'}(t, n)$ holds.

Lemma 1 (Least Interpretation Exists). *For a closure Π_{clo} and stratification $\mathcal{O}$ there is a unique least interpretation $lm(\Pi_{clo})$. Also, $\{\ t \mid lm(\Pi_{clo}) \models K(t, i)\ \}$ is the same for all choices of $\mathcal{O}$.*

5 Concrete and Symbolic Execution

In this section we give the execution semantics of closures, and then generalize to symbolic execution. Symbolic execution yields an algorithm for solving open-world queries assuming an upper-bound on the size of the closure is known. $K^{lm(\Pi_{clo})}$ can be computed by applying rules until reaching a fixpoint. This execution semantics is a simple fixpoint procedure that completely computes the fixpoint at stratum n before applying rules at $n + 1$. The state of execution is a pair (K, n), where K is the partially computed knowledge relation (with stratum markings). The value n is the stratum currently being executed. Figure 6 shows the transition system for execution. Theorem 1 asserts that these two semantics agree.

Theorem 1 (Semantic Equivalence). *Given Π_{clo} and $\mathcal{O}$. If execution produces a state $(K^{lfp}, max(\mathcal{O}) + 1)$, then $K^{lfp} = K^{lm(\Pi_{clo})}$*

Suppose Π is open and the number of facts required to solve $query(\Pi, g)$ can be established. We use this upper-bound to form a symbolic closure of Π, and apply symbolic execution to capture all possible fixpoints obtainable from all possible choices of symbolic values. The output of symbolic execution is a ternary symbolic fixpoint H. An element of H is a triple (t, φ, n) where t is a term (possibly with symbolic values), φ is a quantifier-free formula (possibly with symbolic values), and n is a stratum number. Let the vector $\boldsymbol{s}$ assign a non-symbolic value to every symbolic value, then the concrete fixpoint obtained from these choices is:

$$fix(H, \boldsymbol{s}) \stackrel{def}{=} \{\ (t[\star\backslash\boldsymbol{s}], n) \mid (t, \varphi, n) \in H \text{ and } \models \varphi[\star\backslash\boldsymbol{s}]\} \tag{5}$$

In other words, an element $t[\star\backslash\boldsymbol{s}]$ is in the concrete fixpoint if $\varphi[\star\backslash\boldsymbol{s}]$ evaluates to true. For instance, the sorting problem asks about one input list, which gives an upper-bound on the facts required to close Π_{Sort}. Our algorithm would perform symbolic execution on the closure:

$$\Pi_{Sort} \cup \{input(\star_1, \star_2, \star_3, \star_4) \leftarrow .\ \}$$

Here are a few elements in the resulting symbolic fixpoint:

$$\left\{ \begin{array}{c} (input(\star_1, \star_2, \star_3, \star_4), true, 0), \\ (trace(\star_1, \star_2, \star_3, \star_4), true, 1), \\ (trace(\star_2, \star_1, \star_3, \star_4), \star_1 > \star_2, 1) \end{array} \right\} \subset H$$

For any choice of $\boldsymbol{s}$ the first two elements are in a concrete fixpoint, but the third is there only if $s_1 > s_2$.

Symbolic execution. Computing H makes use of the safety condition that all universally quantified variables range over K. A partial symbolic fixpoint captures all possible elements up to some point in execution. Thus, H can be used to expand quantifiers as follows[3]: Let $\boldsymbol{h}$ be a vector of elements from H, with j^{th} component (t_j, φ_j, m_j). The translator $\mathcal{E}[\![]\!]$ expands quantifiers using the current state H.

$$\mathcal{E}[\![\forall \boldsymbol{y}.\ \neg\psi, \boldsymbol{x}, \boldsymbol{h}', H]\!] \stackrel{def}{=} \bigwedge_{\boldsymbol{h} \in H^{|\boldsymbol{y}|}} \left(\left[\bigwedge_{y_j} y_j = t_j \right] \Rightarrow \neg \mathcal{E}[\![\psi, \boldsymbol{x}\boldsymbol{y}, \boldsymbol{h}'\boldsymbol{h}, H]\!] \right).$$

$$\mathcal{E}[\![\exists i.\ (\psi \wedge K(x_j, i)), \boldsymbol{x}, \boldsymbol{h}, H]\!] \stackrel{def}{=} \psi[i\backslash m_j] \wedge \varphi_j.$$

Figure 7 gives the symbolic execution procedure. The next theorem asserts that symbolic execution correctly captures the affects of choosing values $\boldsymbol{s}$ for $\star$ on the possible fixpoints.

Theorem 2 (Correctness). *Let $\Pi \cup F$ be a closure of Π with symbolic facts F, and H^{lfp} be any state $(H^{lfp}, max(\mathcal{O}) + 1)$ obtained by symbolically executing this closure. Then, for any choice of $\boldsymbol{s}$ of symbolic values:*

$$K^{lm(\Pi \cup F[\star\backslash\boldsymbol{s}])} = fix(H^{lfp}, \boldsymbol{s}).$$

[3] For simplicity, we assume all variables with the same name are the same variable. If this is not the case, then variables must be renamed beforehand.

Init:	$(\emptyset, min(\mathcal{O}))$	
Step:	$(H, n) \Longrightarrow (H \cup \{(t[\boldsymbol{x}\backslash\boldsymbol{t}], \varphi, n)\}, n)$	If $\begin{cases} R \text{ as in Figure 6}, \boldsymbol{h} = H^{\lvert\boldsymbol{x}\rvert}, \text{ and} \\ \varphi \stackrel{def}{=} \bigwedge_{p_i} \mathcal{E}[\![\mathcal{R}[\![p_i, \boldsymbol{x}, n]\!], \boldsymbol{x}, \boldsymbol{h}, H]\!] \wedge \\ \quad\quad \bigwedge_j (x_j = t_j) \end{cases}$
Fix:	$(H, n) \Longrightarrow (H, n+1)$	If steps are subsumed by elements in H

Fig. 7. Symbolic Execution

Given $query(\Pi, g)$ and a symbolic-upper bound F, then compute H^{lfp} for $\Pi \cup F \cup \{c_{fresh} \leftarrow g.\}$. Let the set of goal constraints be $G \stackrel{def}{=} \{\, \varphi \mid (c_{fresh}, \varphi, n) \in H^{lfp} \,\}$. By Theorem 2 all concrete closures obtainable from F satisfying the goal are characterized by the solutions to $\bigvee_{\varphi \in G} \varphi$.

6 Discussion and Conclusion

In general, open-world queries are undecidable and the techniques presented here are incomplete. Incompleteness can manifest in two ways: (1) Symbolic execution may not terminate for a single symbolic closure. This can happen if an OLP encodes an infinite state transition system. For some such systems symbolic execution cannot summarize all behaviors within a finite number of steps. (2) It may be impossible to determine an upper-bound on the size of the closure. This can happen when an OLP places non-linear cardinality constraints on the size of the closure. Determining an upper-bound amounts to deciding a system of non-linear Diophantine inequalities.

FORMULA addresses these issues in two ways: Transition systems can be recognized, and then full computation of the symbolic fixpoint can be avoided. This helps specifically for the transition system scenario where bounded unrolling may be sufficient. FORMULA extracts and presolves cardinality constraints to obtain reasonable estimates for closure size. If a guess is unsatisfiable, then it tries again with a larger guess; this process may never terminate. The fixpoint procedures presented here are not the most efficient. A realistic engine needs to optimize them. For both the concrete and symbolic cases, FORMULA combines static analysis, eager simplification, and efficient indexing to avoid unnecessary execution of rules. Subsumption tests are performed infrequently at the risk of extra unnecessary computation.

In conclusion, open-world logic programming is a powerful approach for formal specifications that also integrates deeply with modern solvers. It has a simple formal semantics, can express a wide range of problems, and can unify a wide range of verification/analysis/synthesis tasks under the single open-world query operation. In this paper we focused our comparisons on other logic programming paradigms. See [30] for a less rigorous comparison with the relational model finder *Alloy* [31] and the term rewriting system *Maude* [32].

References

1. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: FASE. (2009) 18–33
2. Büttner, F., Egea, M., Cabot, J.: On verifying atl transformations using 'off-the-shelf' smt solvers. In: MoDELS. (2012) 432–448
3. Maoz, S., Ringert, J.O., Rumpe, B.: Cd2alloy: Class diagrams analysis using alloy revisited. In: MoDELS. (2011) 592–607
4. Varró, D.: Automated formal verification of visual modeling languages by model checking. Software and System Modeling **3**(2) (2004) 85–113
5. Ball, T., Bjørner, N., de Moura, L.M., McMillan, K.L., Veanes, M.: Beyond first-order satisfaction: Fixed points, interpolants, automata and polynomials. In: SPIN. (2012) 1–6
6. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S., Zufferey, D.: P: Safe Asynchronous Event-Driven Programming. In: PLDI. (2013)
7. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about Metamodeling with Formal Specifications and Automatic Proofs. In: MoDELS. (2011) 653–667
8. Jackson, E.K., Kang, E., Dahlweid, M., Seifert, D., Santen, T.: Components, Platforms and Possibilities: Towards Generic Automation for MDA. In: EMSOFT. (2010) 39–48
9. Gelder, A.V., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38**(3) (1991) 620–650
10. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.: Boom analytics: exploring data-centric, declarative programming for the cloud. In: EuroSys. (2010) 223–236
11. Becker, M.Y., Russo, A., Sultana, N.: Foundations of logic-based trust management. In: IEEE Symposium on Security and Privacy. (2012) 161–175
12. Li, N., Mitchell, J.C.: Datalog with constraints: A foundation for trust management languages. In: PADL. (2003) 58–73
13. Schäfer, M., de Moor, O.: Type inference for datalog with complex type hierarchies. In: POPL. (2010) 145–156
14. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: TACAS. (2008) 337–340
15. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. **19/20** (1994) 503–581
16. Holzbaur, C., Menezes, F., Barahona, P.: Defeasibility in clp(q) through generalized slack variables. In: CP. (1996) 209–223
17. Frühwirth, T.W.: Constraint Handling Rules. In: Constraint Programming. (1994) 90–107
18. Hermenegildo, M.V., Bueno, F., Carro, M., López-García, P., Morales, J.F., Puebla, G.: An overview of the ciao multiparadigm language and program development environment and its design philosophy. In: Concurrency, Graphs and Models. (2008) 209–237
19. Apt, K.R., Wallace, M.: Constraint logic programming using Eclipse. Cambridge University Press (2007)
20. Dovier, A., Formisano, A., Pontelli, E.: An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. J. Exp. Theor. Artif. Intell. **21**(2) (2009) 79–121
21. Lifschitz, V.: Twelve definitions of a stable model. In: ICLP. (2008) 37–51

22. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp* : A conflict-driven answer set solver. In: LPNMR. (2007) 260–265
23. Syrjänen, T., Niemelä, I.: The Smodels System. In: LPNMR. (2001) 434–438
24. Gebser, M., Schaub, T., Thiele, S.: Gringo : A new grounder for answer set programming. In: LPNMR. (2007) 266–271
25. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In: ICLP. (2005) 52–66
26. Mellarkod, V.S., Gelfond, M.: Integrating answer set reasoning with constraint solving techniques. In: FLOPS. (2008) 15–31
27. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: ICLP. (2009) 235–249
28. Frühwirth, T.W., Shapiro, E.Y., Vardi, M.Y., Yardeni, E.: Logic programs as types for logic programs. In: LICS. (1991) 300–309
29. Jackson, E.K., Schulte, W., Bjørner, N.: Detecting specification errors in declarative languages with constraints. In: MoDELS. (2012) 399–414
30. Jackson, E.K., Schulte, W.: Understanding specification languages through their model theory. In: Monterey Workshop. (2012) 396–415
31. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: TACAS. (2007) 632–647
32. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285**(2) (2002) 187–243