Single-Shot Epistemic Logic Program Solving

Manuel Bichler, Michael Morak, Stefan Woltran

TU Wien, Vienna, Austria

(surname)@dbai.tuwien.ac.at

Abstract

Epistemic Logic Programs (ELPs) are an extension of Answer Set Programming (ASP) with epistemic operators that allow for a form of meta-reasoning, that is, reasoning over multiple possible worlds. Existing ELP solving approaches generally rely on making multiple calls to an ASP solver in order to evaluate the ELP. However, in this paper, we show that there also exists a direct translation from ELPs into non-ground ASP with bounded arity. The resulting ASP program can thus be solved in a single shot. We then implement this encoding method, using recently proposed techniques to handle large, non-ground ASP rules, into a prototype ELP solving system. This solver exhibits competitive performance on a set of ELP benchmark instances.

1 Introduction

Epistemic Logic Programs (ELPs) are an extension of the well-established formalism of Answer Set Programming (ASP), a generic, fully declarative logic programming language that allows us to encode problems in such a way that the resulting answers (called answer sets) directly correspond to solutions of the encoded problem [Brewka et al., 2011]. Negation in ASP is generally interpreted according to the stable model semantics [Gelfond and Lifschitz, 1988], that is, as negation-as-failure, also called default negation. The default negation \( \neg a \) of an atom \( a \) is true if there is no justification for \( a \) in the same answer set, making it a “local” operator in the sense that it is defined relative to the same answer set. ELPs, on the other hand, extend ASP with the epistemic negation operator \( \text{not} \) that allows for a form of meta-reasoning, that is, reasoning over multiple answer sets. Intuitively, an epistemically negated atom \( \text{not} a \) expresses that \( a \) cannot be proven true, that is, it is false in at least one answer set. Thus, epistemic negation is defined relative to a collection of answer sets, referred to as a world view. The main reasoning task for ELPs, namely checking that a world view exists, is \( \Sigma^P_2 \)-complete [Shen and Eiter, 2016], one level higher on the polynomial hierarchy than solving ground ASP programs [Eiter and Gottlob, 1995].

Michael Gelfond [1991; 1994] recognized epistemic negation as a desired construct for ASP early on and introduced the model operators \( K \) (“known” or “provably true”) and \( M \) (“possible” or “not provably false”) to address this. \( K a \) and \( M a \) stand for \( \neg \neg a \) and \( \neg \neg \neg a \), respectively.

Example 1. A classical example for the use of epistemic negation is the presumption of innocence rule

\[
\text{innocent}(X) \leftarrow \text{not guilty}(X),
\]

namely: a person is innocent if they cannot be proven guilty.

Renewed interest in recent years has revealed several flaws in the original semantics, and several approaches (cf. e.g. [Gelfond, 2011; Truszczynski, 2011; Kahl, 2014; del Cerro et al., 2015; Shen and Eiter, 2016]) aimed to refine them in such a way that unintended world views are eliminated. In this work, we will settle on the semantics proposed by Shen and Eiter [2016]. The flurry of new research also led to the development of ELP solving systems [Kahl et al., 2015; Son et al., 2017]. Such solvers employ readily available, highly efficient ASP systems like clingo [Gebser et al., 2012a; 2014] and WASP [Alviano et al., 2013], especially making use of the former solver’s multi-shot solving functionality. However, these ELP solving systems rely on ground ASP programs when calling the ASP solver, which, for complexity-theoretic reasons, generally requires multiple calls in order to check for world view existence.

Contributions. Our contributions in this paper are twofold:

\( \triangleright \) Firstly, we propose a novel translation from ELPs to ASP programs using large non-ground ASP rules, such that the ELP can be solved by an ASP solving system in a single shot. This is done via a recently proposed encoding technique [Bichler et al., 2016b] that uses large ASP rules to formulate complex checks. This technique builds on a result by Eiter et al. [2007] that states that evaluating non-ground ASP programs with bounded predicate arity is \( \Sigma^P_2 \)-complete, which matches the complexity of evaluating ELPs. Our proposed translation is therefore optimal from a complexity-theoretic point of view. From a practical point of view, such an encoding avoids multiple calls to the ASP solver. State-of-the-art systems use sophisticated heuristics and learning, and multiple calls might result in a loss of knowledge about the problem instance, which the solver has already learned.

We further discuss how our encoding needs to be constructed in order to be useful in practice. In particular, in current ASP systems, non-ground ASP programs first need
to be grounded, that is, all variables need to be replaced by all allowed combinations of constants. Since our encoding makes use of large non-ground rules, a naive grounding will often not terminate, since there may be hundreds or thousands of variables in a rule. However, as proposed in [Bichler et al., 2016b], we make use of the lprop rule decomposition tool [Bichler et al., 2016a] that splits such large rules into smaller ones that are more easily grounded, by making use of the concept of treewidth and tree decompositions [Bodlaender, 1993]. To use this tool to its full potential, the large rules we use in our encoding must be constructed carefully, in order for lprop to split them up optimally.

Secondly, we propose a prototype implementation of our ELP-to-ASP rewriting approach and combine it with the state-of-the-art ASP solving system clingo [Gebser et al., 2014] in order to evaluate its performance. We compare our system against EP-ASP [Son et al., 2017] on different benchmarks found in the literature. Our system shows competitive performance on these benchmarks, in particular on instances with good structural properties.

2 Preliminaries

Answer Set Programming (ASP). A ground logic program (also called answer set program, ASP program, or simply program) is a pair \( \Pi = (A, R) \), where \( A \) is a set of propositional (i.e. ground) atoms and \( R \) is a set of rules of the form

\[
a_1 \lor \cdots \lor a_t \leftarrow a_{t+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n; \quad (1)
\]

where the comma symbol stands for conjunction, \( n \geq m \geq t \) and \( a_i \in A \) for all \( 1 \leq i \leq n \). Each rule \( r \in R \) of form (1) consists of a head \( H(r) = \{a_1, \ldots, a_t\} \) and a body given by \( B^+(r) = \{a_{t+1}, \ldots, a_m\} \) and \( B^-(r) = \{a_{m+1}, \ldots, a_n\} \). A literal \( \ell \) is either an atom \( a \) or its (default) negation \( \neg a \). A literal \( \ell \) is true in a set of atoms \( I \subseteq A \) if \( \ell = a \) and \( a \in I \), or \( \ell = \neg a \) and \( a \notin I \); otherwise \( \ell \) is false in \( I \). A set \( M \subseteq A \) is called a model of \( r \) if \( B^+(r) \cap M \subseteq \) \( \emptyset \) and \( B^-(r) \cap M \neq \emptyset \). We denote the set of models of \( r \) by \( \text{models}(r) \) and the models of a logic program \( \Pi = (A, R) \) are given by \( \text{models}(\Pi) = \bigcap_{r \in R} \text{models}(r) \). The GL-reduct \( \Pi^G \) of a ground logic program \( \Pi \) with respect to a set of atoms \( I \subseteq A \) is the program \( \Pi^G = (A, \{H(r) \leftarrow B^+(r) \mid r \in R, B^-(r) \cap I = \emptyset\}) \).

Definition 2. [Gelfond and Lifschitz, 1988] \( M \subseteq A \) is an answer set of a program \( \Pi \) if (1) \( M \in \text{models}(\Pi) \) and (2) there is no subset \( N \subset M \) such that \( N \in \text{models}(\Pi^M) \).

The set of answer sets of a program \( \Pi \) is denoted \( \text{AS}(\Pi) \). The consistency problem of ASP (decide whether for given \( \Pi, \text{AS}(\Pi) \neq \emptyset \)) is \( \Sigma^p_3 \)-complete [Eiter and Gottlob, 1995].

General non-ground logic programs differ from ground logic programs in that variables may occur in rules. Such rules are \( \forall \)-quantified first-order implications of the form \( H_1 \lor \cdots \lor H_k \leftarrow P_1, \ldots, P_m, \neg N_1, \ldots, \neg N_m \) where \( H_i, P_i \) and \( N_i \) are (non-ground) atoms. A non-ground atom \( A \) is of the form \( p(X, c) \) and consists of a predicate name \( p \), as well as a sequence of variables \( X \) and constants \( c \) from a domain \( \Delta \), with \( |X| + |c| \) being the arity of \( p \). Let \( \text{var}(A) \) denote the set of variables \( X \) in a non-ground atom \( A \). This notation naturally extends to sets. We will denote variables by capital letters, constants and predicates by lower-case words. A non-ground rule can be seen as an abbreviation for all possible instantiations of the variables with domain elements from \( \Delta \). This step is usually explicitly performed by a grounder that transforms a (non-ground) logic program into a set of ground rules of form (1). Note that in general, such a ground program can be exponential in the size of the non-ground program. However, for non-ground programs of bounded arity, the consistency problem is \( \Sigma^p_3 \)-complete [Eiter et al., 2007].

Epistemic Logic Programs. A ground epistemic logic program (ELP) is a pair \( \Pi = (A, R) \), where \( A \) is a set of propositional atoms and \( R \) is a set of rules of the following form:

\[
a_1 \lor \cdots \lor a_k \leftarrow \ell_1, \ldots, \ell_m, 
\]

where each \( a_i \) is an atom, each \( \ell_i \) is a literal, and each \( \xi_i \) is an epistemic literal, that is, a formula \( \neg \ell \), where \( \neg \) is the epistemic negation operator, and \( \ell \) is a literal. W.l.o.g. we assume that no atom appears twice in a rule. Let \( \text{edit}(r) \) denote the set of all epistemic literals occurring in a rule \( r \in R \). This notation naturally extends to programs. Let \( H(r) = \{a_1, \ldots, a_k\} \), and let \( B(r) = \{\ell_1, \ldots, \ell_m, \xi_1, \ldots, \xi_j, \neg \xi_{j+1}, \ldots, \neg \xi_n\} \), that is, the set of elements appearing in the rule body.

In order to define the main reasoning tasks for ELPs, we recall the notion of the epistemic reduct [Shen and Eiter, 2016]. Let \( \Phi \subseteq \text{edit}(\Pi) \) (called a guess). The epistemic reduct \( \Pi^\Phi \) of the program \( \Pi = (A, R) \) w.r.t. \( \Phi \) consists of the rules \( r^- \cap r \in \emptyset \) \( \text{edit}(\Pi) \) \( \emptyset \) replaced by \( \emptyset \) (resp. \( \emptyset \)). Note that \( \Pi^\Phi \) is a logic program without epistemic negation. This leads to the following, central definition.

Definition 3. Let \( \Phi \) be a guess. The set \( M = \text{AS}(\Pi^\Phi) \) is called a candidate world view of \( \Pi \) if

1. \( M \neq \emptyset \),
2. for each epistemic literal \( \neg \ell \in \Phi \), there exists an answer set \( M \in \mathcal{M} \) wherein \( \ell \) is false, and
3. for each epistemic literal \( \neg \ell \in \text{edit}(\Pi) \setminus \Phi \), it holds that \( \ell \) is true in each answer set \( M \in \mathcal{M} \).

Example 4. Let \( \Pi \) be the following ELP, with \( R = \{r_1, r_2\} \):

\[
\begin{align*}
r_1: p & \leftarrow \neg q \\
r_2: q & \leftarrow \neg p
\end{align*}
\]

ELP \( \Pi \) has two candidate world views: (1) \( \Phi = \{\neg q\} \) with \( \text{AS}(\Pi^\Phi) = \{ \{p\} \} \); (2) \( \Phi = \{\neg p\} \) with \( \text{AS}(\Pi^\Phi) = \{ \{q\} \} \).

The main reasoning task we treat in this paper is the ELP view existence problem (or ELP consistency), that is, given an ELP \( \Pi \), decide whether a candidate world view exists. This problem is known to be \( \Sigma^p_3 \)-complete [Shen and Eiter, 2016].

Tree Decompositions. A tree decomposition of a graph \( G = (V, E) \) is a pair \( T = (T, \chi) \), where \( T \) is a rooted tree and \( \chi \) is a labelling function over nodes \( t \), with \( \chi(t) \subseteq V \), such that the following holds: (i) for each \( v \in V \) there is a node \( t \in T \)
such that \( v \in \chi(t) \); (ii) for each \( \{v, w\} \in E \) there is a node \( t \) in \( T \) such that \( \{v, w\} \subseteq \chi(t) \); and (iii) for all nodes \( r, s, t \) in \( T \), where \( s \) lies on the path from \( r \) to \( t \), \( \chi(r) \cap \chi(t) \subseteq \chi(s) \).

The width of a tree decomposition is defined as the maximum cardinality of \( \chi(t) \) minus one. The treewidth of a graph \( G \) is the minimum width over all tree decompositions of \( G \). Trees have treewidth 1, cliques of size \( k \) have treewidth \( k \). Finding a tree decomposition of minimal width is NP-hard in general.

### 3 Single-Shot ELP Solving

In this section, we provide our novel translation for solving ELPs via a single call to an ASP solving system. The goal is to transform a ground ELP \( \Pi \) to a non-ground ASP program \( \Pi' \) with predicates of bounded arity, such that \( \Pi \) is consistent (i.e., it has a candidate world view) iff \( \Pi' \) has at least one answer set. A standard ASP solver can then decide the consistency problem for the ELP \( \Pi \) in a single call, by solving \( \Pi' \).

#### 3.1 Reducing ELPs to ASP Programs

The reduction is based on an encoding technique proposed in Bichler et al. (2016b), which uses large, non-ground rules. Given an ELP \( \Pi \), the ASP program \( \Pi' \) will roughly be constructed as follows. \( \Pi' \) contains a guess part that chooses a set of epistemic literals from \( \text{elit}(\Pi) \), representing a guess \( \Phi \) for \( \Pi \). Then, the check part verifies that, for \( \Phi \), a candidate world exists. In all, the ASP program \( \Pi' \) consists of five parts:

\[
\Pi' = \Pi'_{\text{facts}} \cup \Pi'_{\text{guess}} \cup \Pi'_{\text{check1}} \cup \Pi'_{\text{check2}} \cup \Pi'_{\text{check3}},
\]

where the sub-program \( \Pi'_{\text{facts}} \) is a set of facts representing the ELP \( \Pi \), and \( \Pi'_{\text{check1}} \) represents the part of the program that checks Condition 1 of Definition 3. We now proceed to the construction of the program \( \Pi' \). Let \( \Pi = (A, \mathcal{R}) \) be the ELP to reduce from. To ease notation, let \( A = \{a_1, \ldots, a_n\} \).

The set of facts \( \Pi'_{\text{facts}} \). \( \Pi'_{\text{facts}} \) represents basic knowledge about the ELP \( \Pi \), plus some auxiliary facts, and is given as:

- \( \text{atom}(a) \), for each atom \( a \in A \);
- \( \text{elit}(\ell) \), for each epistemic literal \( \ell \in \text{elit}(\Pi) \);
- \( \text{leq}(0, 0) \), \( \text{leq}(0, 1) \), and \( \text{leq}(1, 1) \), representing the less or equal relation for boolean values; and
- \( \text{or}(0, 0) \), \( \text{or}(0, 1) \), \( \text{or}(1, 0) \), and \( \text{or}(1, 1) \), representing the boolean relation or.

Sub-Program \( \Pi'_{\text{guess}} \). This part of the program consists of a single, non-ground rule that guesses a subset of the epistemic literals (stored in predicate \( g \)) as follows:

\[
g(L, 1) \vee g(L, 0) \leftarrow \text{elit}(L).
\]

Shorthands. Before defining the three check parts of the program, we will introduce some useful shorthands which will be used at several occasions. To this end, we use a context identifier \( C \). We first define the following:

\[
H_{\text{sat}}^C(A) \equiv v_C(A, 1) \vee v_C(A, 0),
\]

that is, \( H_{\text{sat}}^C(A) \) guesses a truth assignment for some variable \( A \) and stores it in relation \( v_C \). We will often use variables \( X = \{X_1, \ldots, X_n\} \) or \( Y = \{Y_1, \ldots, Y_n\} \) to represent a subset \( M \) of \( A \), where assigning \( X_i \) to 1 characterizes \( a_i \in M \), and \( X_i = 0 \) otherwise. Let

\[
B_{\text{sat}}^C(X) \equiv \bigwedge_{a_i \in A} v_C(a_i, X_i),
\]

that is, \( B_{\text{sat}}^C(X) \) extracts the truth assignment from relation \( v_C \) into the variables \( X \) as described above. Finally, for some rule \( r \) in \( \Pi \), we define a formula \( B_{\text{sat}}(X, Y, S) \) that checks whether the rule \( r \) is satisfied in the epistemic reduct \( \Pi^\Phi \) w.r.t. the guess \( \Phi \) encoded in the relation \( g \), when the negative body (resp. positive body and head) is evaluated over the set of atoms encoded by \( X \) (resp. \( Y \)). If the rule is satisfied, \( B_{\text{sat}}(X, Y, 1) \) should hold, and \( B_{\text{sat}}(X, Y, 0) \) otherwise. This is done as follows. Let \( r \) contain the variables \( \{a_{i_1}, \ldots, a_{i_m}\} \) (recall that no atom appears twice in a rule), where \( i_1, i_2, \ldots, i_m \in \{1, \ldots, n\} \). For ease of notation, we will use a four-ary or relation, which can easily be split into two of our three-ary or atoms using a helper variable \( T \):

\[
or(W, X, Y, Z) \leftrightarrow or(W, X, T), or(T, Y, Z).
\]

The following is the central building block of our reduction:

\[
B_{\text{sat}}(X, Y, R_m) \equiv R_0 = 0, \bigwedge_{a_{i_j} \in B(r)} \text{or}(R_{j-1}, 1-Y_j, R_j), \bigwedge_{a_{i_j} \in B(r)} \text{or}(R_{j-1}, X_j, R_j), \bigwedge_{a_{i_j} \in B(r)} g(a_{i_j}, N_j), or(N_j, 1-X_j, T_j), or(R_{j-1}, 1-T_j, R_j), \bigwedge_{\neg a_{i_j} \in B(r)} g(\neg a_{i_j}, N_j), or(N_j, Y_j, T_j),\bigwedge_{\neg a_{i_j} \in B(r)} \bigwedge_{\neg a_{i_j} \in B(r)} g(\neg a_{i_j}, N_j), or(R_{j-1}, N_j, 1-Y_j, R_j), \bigwedge_{\neg a_{i_j} \in B(r)} g(\neg a_{i_j}, N_j), or(R_{j-1}, N_j, X_j, R_j), \bigwedge_{\neg a_{i_j} \in B(r)} g(\neg a_{i_j}, N_j), or(R_{j-1}, N_j, X_j, R_j).
\]

For a rule \( r \), each big conjunction in the above formula encodes a reason for \( r \) to be satisfied. For example, the fifth line encodes the fact that rule \( r \) is true if the disjunct \( \neg a_{i_j} \) is not satisfied, that is, if the epistemic literal \( \neg a_{i_j} \) is not part of the guess \( \Phi \) or the atom \( a_{i_j} \) is false (represented by \( 1-Y_j \)). Each disjunct of rule \( r \) is evaluated in this way, and the results are connected via the or relation. Therefore, \( R_m \) will be 1 if \( r \) is satisfied, and 0 otherwise, as desired. Finally, we define \( B_{\text{sat}}(X, Y) \), which makes sure that the variables \( Y \) identify a strict subset of the atoms identified by \( X \). Let \( B_{\text{sat}}(X, Y) \equiv \bigwedge_{a_i \in A} \text{leq}(Y_i, X_i), or(N_{i-1}, X_{i-1}, Y_i, N_i) \),

We can now proceed with the remainder of our reduction.
Sub-Program $Π_\text{check}_1$. This part of the program needs to check that, given the guess $Φ$ made in $Π_{\text{guess}}$, there exists at least one answer set of the epistemic reduct $Π^Φ$, as per Definition 3(1). Therefore, according to Definition 2, we need to find a set $M \subseteq A$, such that (1) $M$ is a model of $Π^Φ$, and (2) there is no proper subset of $M$ that is a model of the GL-reduct $(Π^Φ)^M$. $Π_\text{check}_1$ contains the following rules:

- $H_{\text{val}}(A) \leftarrow \text{atom}(A)$;
- $⊥ \leftarrow B_{\text{val}}(X), B^r_{\text{sat}}(X, X, 0)$, for each $r \in R$; and
- $⊥ \leftarrow B^r_{\text{red}}$.

The first rule guesses a truth assignment for all atoms. The second rule verifies that there is no rule in $Π^Φ$ that is violated by the candidate answer set $M$, represented by the variables $X$, guessed by the first rule. $B^r_{\text{red}}$ checks whether a subset of $M$ is a model of the GL-reduct $(Π^Φ)^M$. To this end, let

$$B^r_{\text{red}} \equiv B^r_{\text{val}}(X), B_{\text{sat}}(X, Y), \bigwedge_{r \in R} B^r_{\text{sat}}(X, Y, 1).$$

The last big conjunction in $B^r_{\text{red}}$ guarantees that the subsequent $N \subseteq M$ identified by the variables $Y$ is indeed a model of every rule in the GL-reduct $(Π^Φ)^M$. This completes $Π_\text{check}_1$.

Sub-Program $Π_\text{check}_2$. This part needs to check that, for every epistemic literal $\text{not} \ell \in Φ$, the epistemic reduct $Π^Φ$ has some answer set wherein $\ell$ is false. $Π_\text{check}_2$ contains the following rules and facts, for each epistemic literal $\text{not} \ell \in \text{elit}(Π)$ (used as the context $C$ so guesses are independent):

- $H_{\text{val}}(A) \leftarrow \text{atom}(A), g(\ell, 1)$;
- $v(\ell, η)$, where $η = 1$ if $\ell = a$, or $η = 0$ if $\ell = a$;
- $⊥ \leftarrow B^r_{\text{val}}(X), B^r_{\text{sat}}(X, X, 0)$, for each $r \in R$; and
- $⊥ \leftarrow B^r_{\text{red}}$.

These rules guess, for each epistemic literal $\text{not} \ell \in Φ$, a candidate answer set $M$ wherein $\ell$ is false, and then verify that $M$ is indeed an answer set, using the same technique as in $Π_\text{check}_1$. This ensures Condition 2 of Definition 3.

Sub-Program $Π_\text{check}_3$. Finally, this part needs to check that, for every epistemic literal $\text{not} \ell \in \text{elit}(Π) \setminus Φ$, every answer set of $Π^Φ$ satisfies $\ell$. The construction makes use of the technique of saturation [Eiter and Gottlob, 1995]:

- $H_{\text{val}}(A) \leftarrow \text{atom}(A)$;
- $v_{\text{check}}_j(A, 0) \leftarrow \text{sat}, \text{atom}(A)$;
- $v_{\text{check}}_j(A, 1) \leftarrow \text{sat}, \text{atom}(A)$; and
- $⊥ \leftarrow \text{not} \text{sat}$.

This setup checks that, for every candidate answer set $M$ guessed in the first rule, the atom $\text{sat}$ is derived. Since we are only interested in answer sets, we first check that $M$ is indeed one, using the following rules, similarly to $Π_\text{check}_1$:

- $\text{sat} \leftarrow B^r_{\text{val}}(X), B^r_{\text{sat}}(X, X, 0)$, for each $r \in R$; and
- $\text{sat} \leftarrow B^r_{\text{check}}$.

It now remains to check that in each answer set $M$ (that is, where $\text{sat}$ has not been derived yet) all epistemic literals $\text{not} \ell$ are either in $Φ$, or otherwise $\ell$ is true in $M$. This is done by adding the following rule to $Π_\text{check}_3$:

$$\text{sat} \leftarrow \bigwedge_{\ell \text{ not} \in \text{elit}(Π)} g(a, N_a), v_{\text{check}}_j(a, X_a), \text{or}(N_a, X_a, 1),$$

$$\bigwedge_{\ell \text{ not} \in \text{elit}(Π)} g(\neg a, N_a), v_{\text{check}}_k(a, X_a), \text{or}(N_a, 1-X_a, 1).$$

This completes the reduction. The following result states that the above reduction is polynomial and correct.

**Theorem 5.** Given an ELP $Π$, the reduction above runs in time $O(e n)$, where $n$ is the size of $Π$ and $e = |\text{elit}(Π)|$, uses predicates of arity at most three, and is correct.

**Proof.** (Idea) Correctness follows by construction; it can be verified that each of the three parts of the constructed program $Π'$ indeed ensures precisely the conditions that define a candidate world view. Each answer set $A$ of $Π'$ is a witness for the fact that a guess $Φ \subseteq \text{elit}(Π)$ encoded in $A$ indeed gives rise to a candidate world view. Predicates of arity at most three are used if the four-ary or relation is not materialized as an actual relation in ASP, but viewed as a shorthand for two connected ternary or relations. The reduction’s runtime (and output size) can be shown to be in $O(e n)$ by noting the fact that the construct $B_{\text{red}}$ is of size linear in $n$ (precisely encodes each rule using the $\text{or}$ predicates). $B_{\text{red}}$ is then use once for each epistemic literal in $Π$ (cf. check 2).

Note that the above theorem shows that our reduction is indeed worst-case optimal as claimed in Section 1: checking consistency of non-ground, fixed-arity ASP programs is $\Sigma_P^P$-complete, as is checking world-view existence for ELPs.

**Example 6.** Recall program $Π = (A, R)$ from Example 4. Let $A = \{a_1, a_2\}$, where $a_1 = p$ and $a_2 = q$. Let rule $r_3 \in R$ contain the atoms $\{a_1, a_2\}$, where $i_1 = 2$ and $i_2 = 1$. The core construct in our construction is $B_{\text{sat}}(\cdot, \cdot, \cdot)$. For rule $r_3$, we have

$$B^r_{\text{sat}}(X_1, Y_1, Y_2, R_3) \equiv \text{or}(0, Y_2, R_3), g(p, N_2), \text{or}(N_2, 1-X_1, T_2), \text{or}(R_1, 1-T_2, R_2).$$

### 3.2 Using the Reduction in Practice

As we have seen, using the construction in the previous subsection, we can solve the consistency problem for a given ELP via a single call to an ASP solving system. However, when trying this in practice, the performance is less than optimal, mainly for the following reason. At several points in the construction, large non-ground rules are used (i.e. where $B_{\text{red}}$ appears in a rule body). As noted in Section 2, these rules need to be grounded, but may contain hundreds or thousands of variables, which need to be replaced by all possible combinations of constants; a hopeless task for ASP grounders.

However, as noted in [Bichler et al., 2016b], such large rules can often be decomposed into smaller, more manageable rules, using the lpop tool [Bichler et al., 2016a]. This tool roughly works as follows: (1) compute a rule graph $G_r$ for each non-ground rule $r$, where there is a vertex for each variable $V$ in $r$, and there is an edge between $V_1$ and $V_2$. 


if the two variables appear together in an atom of \( r \); then (2) compute a tree decomposition of \( G_r \) of minimal width; and finally, (3) in a bottom-up manner, output a rule for each node in the tree decomposition. The resulting rules each contain only as many variables as the treewidth of \( G_r \) (plus one), and, together, are equivalent to the original rule \( r \). After this rule decomposition step, grounding now becomes much easier, since the number of variables in each rule is reduced. Note that, since finding optimal tree decompositions is NP-hard, \( l po pt \) employs heuristics to find good decompositions.

In our construction, \( B^c_{\text{red}} \) stands for a long rule body that basically encodes the entire input ELP \( \Pi \). Each atom \( a_i \) in \( \Pi \) is represented by the two variables \( X_i \) and \( Y_i \). If we represent \( \Pi \) as a graph \( G_{\Pi} \), where each atom \( a_i \) is a vertex, and there is an edge between two atoms if they appear together in a rule in \( \Pi \), then this graph structure can be found (as a minor) in the rule graph of \( B^c_{\text{red}} \). However, in addition, \( B^c_{\text{red}} \) also adds a series of \( \text{or}(\cdot,\cdot,\cdot) \)-atoms (via \( B_{ss}(X,Y) \)), that introduce additional connections in the rule graph of \( B^c_{\text{red}} \). These connections may increase the treewidth substantially. In fact, even if \( G_{\Pi} \) has a treewidth of 1, by introducing the additional connections in a bad way, the treewidth may increase arbitrarily: imagine that \( G_{\Pi} \) is a chain, depicted in black in Figure 1, and imagine the \( \text{or}(\cdot,\cdot,\cdot) \)-chain from \( B_{ss}(X,Y) \) is inserted into \( G_{\Pi} \), illustrated in pink. The treewidth now depends on the chain’s length, and \( l po pt \) can no longer split the rule well.

The solution to this problem is the following. Instead of constructing \( B_{ss}(X,Y) \) as given in the construction above, first perform a tree decomposition on \( G_{\Pi} \). Then, construct \( B_{ss}(X,Y) \) along this tree decomposition in such a way that the \( \text{or}(\cdot,\cdot,\cdot) \)-atoms do not introduce additional edges into \( G_{\Pi} \), therefore preserving the treewidth of \( G_{\Pi} \). With this technique, the treewidth of \( B^c_{\text{red}} \) only depends on the treewidth of \( G_{\Pi} \) (and thus on the actual structure of \( \Pi \)), and not on the size of \( \Pi \). This in turn allows \( l po pt \) to work efficiently.

3.3 Discussion
As we have seen, the reduction proposed above allows us to solve ELPs via a single call to an ASP solving system. However, our encoding also has several other properties, which make it very flexible for use with, for example, different ASP semantics, or harder problems. A brief discussion follows.

Other ASP Semantics. Apart from the original semantics for ASP (called \textit{stable model semantics}, [Gelfond and Lifschitz, 1988; 1991]), several different semantics have been proposed that investigate how to interpret more advanced constructs in ASP, like double negation, aggregates, optimization, and more [Lifschitz et al., 1999; Pearce, 2006; Pelov et al., 2007; Ferraris et al., 2011; Faber et al., 2011; Shen et al., 2014]. Epistemic reducts may contain double negation, and we have opted to use the ELP semantics of [Faber et al., 2011], as used in [Shen and Etter, 2016], to interpret this. The actual interpretation of double negation is encoded in the \( B^c_{\text{red}}(\cdot,\cdot,\cdot) \) shorthand defined in our reduction. This construction is very flexible and can easily be modified to use different ASP semantics (e.g. [Lifschitz et al., 1999]).

4 The \textit{selp} System
We implemented the reduction in the previous section as part of the single shot ELP solving toolbox \textit{selp}, available at goo.gl/U1gmQu. In addition, the toolbox features a grounder for ELPs and a grouping script which groups answer sets of the reduction into candidate world views (allowing for enumeration). The tools are implemented in \textit{python} and depend on the parser generator \textit{LARK}\footnote{https://github.com/erezsh/lark}, the tree decomposition tool \textit{lpo pt} [Bichler et al., 2016a], the tree decomposition tool \textit{htd\_main} [Abseher et al., 2017], and the ASP grounder \textit{gringo} [Gebser et al., 2011].

Input Formats. The \textit{selp} solver reads the \textit{EASP\text{-}not} file format, which is a restriction of the ASP input language of \textit{gringo} to plain ground logic programs as defined in Section 2, extended with the \$\text{not}\$ operator for epistemic negation. This allows us to encode ELPs as defined in Section 2, \textit{selp} also supports \textit{EASP\text{-}KM}, defined by adding the operators \( K\$ and \( M\$ instead of \$\text{not}\$. By allowing variables in body elements, both formats also have a \textit{non\text{-}ground} version. The toolbox offers scripts to translate between the two formats.

Toolbox. We briefly present the main building blocks of \textit{selp}.

- \textit{easpGrounder.py} takes as input a \textit{non\text{-}ground} \textit{EASP\text{-}not} program and outputs its equivalent ground form by rewriting it into an ASP program that the \textit{gringo} grounder can understand and ground. This is done by encoding epistemic negation as predicate names and, after grounding, re-introducing epistemic negation where a placeholder predicate appears. Additionally, \textit{easpGrounder.py} supports arithmetics and the sorts format [Kahl et al., 2015] as input.

- \textit{easp2asp.py} is \textit{selp}'s key component. It takes a ground \textit{EASP\text{-}not} program, and performs the reduction given in Section 3.1, also adhering to the practical considerations presented in Section 3.2, and finally outputs the resulting non-ground logic program in the syntax of \textit{gringo}. Optionally, additional \textit{clasp} directives are generated to allow for enumeration as described in Section 3.3.

- \textit{groupWorldViews.py} takes as input \textit{clasp}'s output in JSON format, groups the answer sets into candidate world views.
views according to their $g(\cdot)$ atoms, and outputs them in a human-readable format.

Usage. Suppose the file problem.easp contains a non-ground ELP encoding of a problem of interest and the file instance.easp contains a problem instance. In order to output all candidate world views, one would use the following command (flags --pas and --project enable projection of answer sets onto relevant predicates only. --n0 tells clasp to compute all answer sets, and --outf=2 to print in JSON format. lpopt is used to decompose long rule bodies. The --sat-prepro=3 flag is recommended by lpopt):

cat problem.easp instance.easp | easpgrounder.py -SELFP | easp2asp.py -pas | lpopt | gringo | clasp --n0 --outf=2 --project --sat-prepro=3 | groupWorldViews.py

5 Experimental Evaluation

We tested our system selp against the state-of-the-art ELP solver, EP-ASP [Son et al., 2017], using three test sets. For every test set, we measured the time it took to solve the consistency problem. For selp, clasp was stopped after finding the first answer set. For EP-ASP, search was terminated after finding the first candidate world view. Note that a single answer set of the selp system is enough to establish consistency of an input ELP. EP-ASP needs to compute a full candidate world view to be able to prove consistency.

Experiments were run on a 2.1GHz AMD Opteron 6272 system with 224 GB of memory. Each process was assigned a maximum of 14 GB of RAM. For EP-ASP, we used the required clingo 4.5.3, since newer versions are incompatible with the solver. For selp, we used clingo 5.2.2, hdt_main 1.2.0, and lpopt 2.2. The time it took EP-ASP to rewrite the input to its own internal format was not measured. EP-ASP was called with the preprocessing option for brave and cautious consequences on, since it always ran faster this way. The selp time is the sum of running times of its components.

Benchmark Instances. We used three types of benchmarks, two coming from the ELP literature and one from the QSAT domain that contains structures of low treewidth.

- Scholarship Eligibility (SE). This set of non-ground ELP programs is shipped together with EP-ASP. Its instances encode the scholarship eligibility problem for 1 to 25 students.
- Yale Shooting (YS). This test set consists of 25 non-ground ELP programs encoding a simple version of the Yale Shooting Problem, a conformant planning problem: the only uncertainty is whether the gun is initially loaded or not, and the only fluents are the gun’s load state and whether the turkey is alive. Instances differ in the time horizon. We follow the ELP encoding by Kahl et al. [2015].
- Tree QBFs (TQ). The hardness proof for ELP consistency [Shen and Eiter, 2016] relies on a reduction from certain QBFs with three quantifier blocks. We apply this reduction to the 14 “Tree” instances of QBFEVAL’16 [Pulina, 2016], available at http://www.qbflib.org/family_detail.php?idFamily=56, splitting each instance’s variables into three random quantifier blocks.

Results. The results for the first two test sets are shown in Figure 2. selp solves all instances from (SE) within 30 seconds, while EP-ASP only solves 17 within the time limit of 8 hours. For (YS), on the other hand, selp is able to solve only 6 instances within the time limit of 30 minutes, whereas EP-ASP can solve 17. Finally, for (TQ), selp can solve 6 of the 14 instances within the time limit of 12 hours, whereas EP-ASP was unable to solve any instances at all.

These results confirm that selp is highly competitive on well-structured problems: in the (SE) instances, knowledge about students is not interrelated, and hence the graph $G_H$ of the ground ELP II consists of one component for each student, thus having constant treewidth. The (TQ) instances keep their constant treewidth thanks to the fact that both the reduction from QBF to ELP and from ELP to non-ground ASP (cf. Section 3.2) preserve the low treewidth of the original QBF instance. Different from selp, EP-ASP is not designed to exploit such structural information of ELPs and, consequently, performs worse than selp in these benchmarks. On the other hand, (YS) contains instances of high treewidth, even though it does not depend on the horizon. EP-ASP is therefore able to outperform selp on such instances. A similar observation can be made for the “Bomb in the Toilet” problem, as benchmarked by Son et al. [2017], which inherently contains a huge clique structure. selp is not designed to solve such instances, and is therefore most suited for ELPs of low treewidth, where it efficiently exploits the problem structure.

6 Conclusions

In this paper, we have seen that ELPs can be encoded into ASP programs using long non-ground rules, such that a single call to an ASP solver is sufficient to evaluate them. A prototype ELP solver implementation, selp, performs particularly well on problems whose internal structure is of low treewidth. A combined solver that either calls selp or another state-of-the-art solver based on the treewidth of the input may therefore lead to even better overall performance.

Another topic for future work is that, under the FLP semantics, checking whether a given atom $a$ is true in all candidate world views with a subset-maximal guess $\Phi$ is known to be $\Sigma_2^p$-complete [Shen and Eiter, 2016]. To solve this problem, advanced optimization features of state-of-the-art ASP solvers could allow us to encode this subset-maximality condition, while leaving the core of our encoding unchanged.
Acknowledgements. This work was funded by the Austrian Science Fund (FWF), grant number Y698.

References


