Single-Shot Epistemic Logic Program Solving

Manuel Bichler, Michael Morak, and **Stefan Woltran**
TU Wien, Vienna, Austria
{bichler,morak,woltran}@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 $\mathbf{not}$ that allows for a form of meta-reasoning, that is, reasoning over multiple answer sets. Intuitively, an epistemically negated atom $\mathbf{not}\, a$ 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 Σ_P^3-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 modal operators $\mathbf{K}$ ("known" or "provably true") and $\mathbf{M}$ ("possible" or "not provably false") to address this. $\mathbf{K}a$ and $\mathbf{M}a$ stand for $\neg\mathbf{not}\, a$ and $\mathbf{not}\, \neg a$, respectively.

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

$$innocent(X) \leftarrow \mathbf{not}\, guilty(X),$$

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:

▷ 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 Σ_P^3-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 *lpopt* 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 *lpopt* 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 $\Pi = (\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ is a set of propositional (i.e. ground) atoms and $\mathcal{R}$ is a set of rules of the form

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n; \qquad (1)$$

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

Definition 2. *[Gelfond and Lifschitz, 1988; 1991] $M \subseteq \mathcal{A}$ is an* answer set *of a program Π if (1) $M \in models(\Pi)$ and (2) there is no subset $N \subset M$ such that $N \in models(\Pi^M)$.*

The set of answer sets of a program Π is denoted $AS(\Pi)$. The *consistency problem* of ASP (decide whether, given Π, $AS(\Pi) \neq \emptyset$) is Σ_P^2-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 $H_1 \vee \cdots \vee H_k \leftarrow P_1, \ldots, P_n, \neg N_1, \ldots, \neg N_m$ where H_i, P_i and N_i are (non-ground) atoms. A *non-ground atom A* is of the form $p(\mathbf{X}, \mathbf{c})$ and consists of a predicate name p, as well as a sequence of variables $\mathbf{X}$ and constants $\mathbf{c}$ from a domain Δ, with $|\mathbf{X}| + |\mathbf{c}|$ being the arity of p. Let $var(A)$ denote the set of variables $\mathbf{X}$ 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 Δ. This step is usually explicitly performed by a *grounder* that transforms a (non-ground) logic program into a set of ground rules of the 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 Σ_P^3-complete [Eiter *et al.*, 2007].

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

$$a_1 \vee \cdots \vee a_k \leftarrow \ell_1, \ldots, \ell_m, \xi_1, \ldots, \xi_j, \neg\xi_{j+1}, \ldots, \neg\xi_n,$$

where each a_i is an atom, each ℓ_i is a *literal*, and each ξ_i is an *epistemic literal*, that is, a formula $\mathbf{not}\, \ell$, where $\mathbf{not}$ is the epistemic negation operator, and ℓ is a literal. W.l.o.g. we assume that no atom appears twice in a rule. Let $elit(r)$ denote the set of all epistemic literals occurring in a rule $r \in \mathcal{R}$. 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 $\Phi \subseteq elit(\Pi)$ (called a *guess*). The *epistemic reduct* Π^Φ of the program $\Pi = (\mathcal{A}, \mathcal{R})$ w.r.t. Φ consists of the rules $\{r^\neg \mid r \in \mathcal{R}\}$, where $r^\neg$ is defined as the rule r with all epistemic literals $\mathbf{not}\, \ell$ in Φ (resp. in $elit(\Pi) \setminus \Phi$) replaced by $\top$ (resp. $\neg\ell$). Note that Π^Φ is a logic program without epistemic negation[1]. This leads to the following, central definition.

Definition 3. *Let Φ be a guess. The set $\mathcal{M} = AS(\Pi^\Phi)$ is called a* candidate world view *of Π iff*

1. *$\mathcal{M} \neq \emptyset$,*
2. *for each epistemic literal $\mathbf{not}\, \ell \in \Phi$, there exists an answer set $M \in \mathcal{M}$ wherein ℓ is false, and*
3. *for each epistemic literal $\mathbf{not}\, \ell \in elit(\Pi) \setminus \Phi$, it holds that ℓ is true in each answer set $M \in \mathcal{M}$.*

Example 4. *Let Π be the following ELP, with $\mathcal{R} = \{r_1, r_2\}$:*

$$r_1 : p \leftarrow \mathbf{not}\, q$$
$$r_2 : q \leftarrow \mathbf{not}\, p$$

ELP Π has two candidate world views: (1) $\Phi = \{\mathbf{not}\, q\}$ with $AS(\Pi^\Phi) = \{\{p\}\}$; (2) $\Phi = \{\mathbf{not}\, p\}$ with $AS(\Pi^\Phi) = \{\{q\}\}$.

The main reasoning task we treat in this paper is the *world view existence problem* (or *ELP consistency*), that is, given an ELP Π, decide whether a candidate world view exists. This problem is known to be Σ_3^P-complete [Shen and Eiter, 2016].

Tree Decompositions. A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$, where T is a rooted tree and χ 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

[1]We interpret double negation according to [Faber *et al.*, 2011].

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, and 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 given ELP Π to a non-ground ASP program Π' with predicates of bounded arity, such that Π is consistent (i.e. it has a candidate world view) iff Π' has at least one answer set. A standard ASP solver can then decide the consistency problem for the ELP Π in a single call, by solving Π'.

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 Π, the ASP program Π' will roughly be constructed as follows. Π' contains a guess part that chooses a set of epistemic literals from $\mathit{elit}(\Pi)$, representing a guess Φ for Π. Then, the check part verifies that, for Φ, a candidate world exists. In all, the ASP program Π' consists of five parts:

$$\Pi' = \Pi'_{facts} \cup \Pi'_{guess} \cup \Pi'_{check_1} \cup \Pi'_{check_2} \cup \Pi'_{check_3},$$

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

The set of facts Π'_{facts}. Π'_{facts} represents basic knowledge-about the ELP Π, plus some auxiliary facts, and is given as:

- $\mathit{atom}(a)$, for each atom $a \in \mathcal{A}$;
- $\mathit{elit}(\ell)$, for each epistemic literal $\mathbf{not}\ \ell \in \mathit{elit}(\Pi)$[2];
- $\mathit{leq}(0,0)$, $\mathit{leq}(0,1)$, and $\mathit{leq}(1,1)$, representing the less or equal relation for Boolean values; and
- $\mathit{or}(0,0,0)$, $\mathit{or}(0,1,1)$, $\mathit{or}(1,0,1)$, and $\mathit{or}(1,1,1)$, representing the Boolean relation *or*.

Sub-Program Π'_{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:

$$g(L, 1) \vee g(L, 0) \leftarrow \mathit{elit}(L).$$

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 $\mathcal{C}$. We first define the following:

$$H^{\mathcal{C}}_{val}(A) \equiv v_{\mathcal{C}}(A, 1) \vee v_{\mathcal{C}}(A, 0),$$

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

$$B^{\mathcal{C}}_{val}(\mathbf{X}) \equiv \bigwedge_{a_i \in \mathcal{A}} v_{\mathcal{C}}(a_i, X_i),$$

that is, $B^{\mathcal{C}}_{val}(\mathbf{X})$ extracts the truth assignment from relation $v_{\mathcal{C}}$ into the variables $\mathbf{X}$ as described above. Finally, for some rule r in Π, we define a formula $B^r_{sat}(\mathbf{X}, \mathbf{Y}, S)$ that checks whether the rule r is satisfied in the epistemic reduct Π^{Φ} w.r.t. the guess Φ encoded in the relation g, when the negative body (resp. positive body and head) is evaluated over the set of atoms encoded by $\mathbf{X}$ (resp. $\mathbf{Y}$). If the rule is satisfied, $B^r_{sat}(\mathbf{X}, \mathbf{Y}, 1)$ should hold, and $B^r_{sat}(\mathbf{X}, \mathbf{Y}, 0)$ 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 $i_1, \ldots, i_m \in \{1, \ldots, n\}$. 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:

$$\mathit{or}(W, X, Y, Z) \leftarrow \mathit{or}(W, X, T), \mathit{or}(T, Y, Z).$$

The following is the central building block of our reduction:

$$\begin{aligned}
B^r_{sat}(\mathbf{X}, \mathbf{Y}, R_m) \equiv R_0 = 0, &\bigwedge_{a_{i_j} \in H(r)} \mathit{or}(R_{j-1}, Y_{i_j}, R_j),\\
\bigwedge_{a_{i_j} \in B(r)} \mathit{or}(R_{j-1}, 1\text{-}Y_{i_j}, R_j), &\bigwedge_{\neg a_{i_j} \in B(r)} \mathit{or}(R_{j-1}, X_{i_j}, R_j),\\
\bigwedge_{\mathbf{not}\, a_{i_j} \in B(r)} g(a_{i_j}, N_j), &\mathit{or}(N_j, 1\text{-}X_{i_j}, T_j), \mathit{or}(R_{j-1}, 1\text{-}T_j, R_j),\\
\bigwedge_{\mathbf{not}\, \neg a_{i_j} \in B(r)} g(\neg a_{i_j}, N_j), &\mathit{or}(N_j, Y_{i_j}, T_j), \mathit{or}(R_{j-1}, 1\text{-}T_j, R_j),\\
\bigwedge_{\neg\mathbf{not}\, a_{i_j} \in B(r)} &g(a_{i_j}, N_j), \mathit{or}(R_{j-1}, N_j, 1\text{-}Y_{i_j}, R_j),\\
\bigwedge_{\neg\mathbf{not}\, \neg a_{i_j} \in B(r)} &g(\neg a_{i_j}, N_j), \mathit{or}(R_{j-1}, N_j, X_{i_j}, R_j).
\end{aligned}$$

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 $\neg\mathbf{not}\ a_{i_j}$ is not satisfied, that is, if the epistemic literal $\mathbf{not}\ a_{i_j}$ is part of the guess Φ, or the atom a_{i_j} is false (represented by $1\text{-}Y_{i_j}$). 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 $B_{ss}(\mathbf{X}, \mathbf{Y})$, which makes sure that the variables $\mathbf{Y}$ identify a strict subset of the atoms identified by $\mathbf{X}$. Let $B_{ss}(\mathbf{X}, \mathbf{Y}) \equiv$

$$N_0 = 0, N_n = 1, \bigwedge_{a_i \in \mathcal{A}} \mathit{leq}(Y_i, X_i), \mathit{or}(N_{i-1}, X_i\text{-}Y_i, N_i).$$

We can now proceed with the remainder of our reduction.

[2]Note that we use the literal ℓ as an ASP constant.

Sub-Program Π'_{check_1}. This part of the program needs to check that, given the guess Φ made in Π'_{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 $M \subseteq \mathcal{A}$, 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 $(\Pi^{\Phi})^M$. Π'_{check_1} contains the following rules:

- $H^{check_1}_{val}(A) \leftarrow atom(A)$;
- $\bot \leftarrow B^{check_1}_{val}(\mathbf{X}), B^r_{sat}(\mathbf{X},\mathbf{X},0)$, for each $r \in \mathcal{R}$; and
- $\bot \leftarrow B^{check_1}_{red}$.

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 $\mathbf{X}$, guessed by the first rule. $B^{\mathcal{C}}_{red}$ checks whether a subset of M is a model of the GL-reduct $(\Pi^{\Phi})^M$. To this end, let

$$B^{\mathcal{C}}_{red} \equiv B^{\mathcal{C}}_{val}(\mathbf{X}), B_{ss}(\mathbf{X},\mathbf{Y}), \bigwedge_{r\in\mathcal{R}} B^r_{sat}(\mathbf{X},\mathbf{Y},1).$$

The last big conjunction in $B^{\mathcal{C}}_{red}$ makes sure that the subset $N \subset M$ identified by the variables $\mathbf{Y}$ is indeed a model of every rule in the GL-reduct $(\Pi^{\Phi})^M$. This completes Π'_{check_1}.

Sub-Program Π'_{check_2}. This part needs to check that, for every epistemic literal $\mathbf{not}\,\ell \in \Phi$, the epistemic reduct Π^{Φ} has some answer set wherein ℓ is false. Π'_{check_2} contains the following rules and facts, for each epistemic literal $\mathbf{not}\,\ell \in elit(\Pi)$ (used as the context $\mathcal{C}$ so guesses are independent):

- $H^{\ell}_{val}(A) \leftarrow atom(A), g(\ell,1)$;
- $v_{\ell}(a,\eta)$, where $\eta = 1$ if $\ell = \neg a$, or $\eta = 0$ if $\ell = a$;
- $\bot \leftarrow B^{\ell}_{val}(\mathbf{X}), B^r_{sat}(\mathbf{X},\mathbf{X},0)$, for each $r \in \mathcal{R}$; and
- $\bot \leftarrow B^{\ell}_{red}$.

These rules guess, for each epistemic literal $\mathbf{not}\,\ell \in \Phi$, a candidate answer set M wherein ℓ is false, and then verify that M is indeed an answer set, using the same technique as in Π'_{check_1}. This ensures Condition 2 of Definition 3.

Sub-Program Π'_{check_3}. Finally, this part needs to check that, for every epistemic literal $\mathbf{not}\,\ell \in elit(\Pi) \setminus \Phi$, every answer set of Π^{Φ} satisfies ℓ. The construction makes use of the technique of saturation [Eiter and Gottlob, 1995]:

- $H^{check_3}_{val}(A) \leftarrow atom(A)$;
- $v_{check_3}(A,0) \leftarrow sat, atom(A)$;
- $v_{check_3}(A,1) \leftarrow sat, atom(A)$; and
- $\bot \leftarrow \neg sat$.

This setup checks that, for every candidate answer set M guessed in the first rule, the atom 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'_{check_1}:

- $sat \leftarrow B^{check_3}_{val}(\mathbf{X}), B^r_{sat}(\mathbf{X},\mathbf{X},0)$, for each $r \in \mathcal{R}$; and
- $sat \leftarrow B^{check_3}_{red}$.

It now remains to check that in each answer set M (that is, where sat has not been derived yet) all epistemic literals $\mathbf{not}\,\ell$ are either in Φ, or otherwise ℓ is true in M. This is done by adding the following rule to Π'_{check_3}:

$$sat \leftarrow \bigwedge_{\mathbf{not}\,a\in elit(\Pi)} g(a,N_a), v_{check_3}(a,X_a), or(N_a,X_a,1),$$
$$\bigwedge_{\mathbf{not}\,\neg a\in elit(\Pi)} g(\neg a,N^{\neg}_a), v_{check_3}(a,X^{\neg}_a), or(N^{\neg}_a,1\text{-}X^{\neg}_a,1).$$

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 $O(e \cdot n)$, where n is the size of Π and $e = |elit(\Pi)|$, 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' ensures precisely one of the three conditions that define a candidate world view. Each answer set A of Π' is a witness for the fact that a guess $\Phi \subseteq elit(\Pi)$ 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 $O(e\cdot n)$ by noting the fact that the construct B_{red} is of size linear in n (it precisely encodes each rule using the or predicates). B_{red} is then used once for each epistemic literal in Π (cf. check 2). $\square$

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 Σ^3_P-complete, as is checking world-view existence for ELPs.

Example 6. *Recall program $\Pi = (\mathcal{A},\mathcal{R})$ from Example 4. Let $\mathcal{A} = \{a_1,a_2\}$, where $a_1 = p$ and $a_2 = q$. Let rule $r_2 \in \mathcal{R}$ contain the atoms $\{a_{i_1},a_{i_2}\}$, where $i_1 = 2$ and $i_2 = 1$. We give the core construct, $B^r_{sat}(\cdot,\cdot,\cdot)$ for rule r_2:*

$$B^{r_2}_{sat}(X_1,X_2,Y_1,Y_2,R_2) \equiv or(0,Y_2,R_1),$$
$$g(p,N_2), or(N_2,1\text{-}X_1,T_2), or(R_1,1\text{-}T_2,R_2).$$

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 $B^{\mathcal{C}}_{red}$ 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 *lpopt* 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

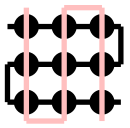


Figure 1: Creating a grid from chains.

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, *lpopt* employs heuristics to find good decompositions.

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

The solution to this problem is the following. Instead of constructing $B_{ss}(\mathbf{X},\mathbf{Y})$ as given in the construction above, first perform a tree decomposition on G_Π. Then, construct $B_{ss}(\mathbf{X},\mathbf{Y})$ along this tree decomposition in such a way that the $or(\cdot,\cdot,\cdot)$ atoms do not introduce additional edges into G_Π, therefore preserving the treewidth of G_Π. With this technique, the treewidth of $B^{\mathcal{C}}_{red}$ only depends on the treewidth of G_Π (and thus on the actual structure of Π), and not on the size of Π. This in turn allows *lpopt* 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 *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 FLP semantics of [Faber *et al.*, 2011], as used in [Shen and Eiter, 2016], to interpret this. The actual interpretation of double negation is encoded in the $B^r_{sat}(\cdot,\cdot)$ 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]).

Enumeration of World Views. Modern ASP systems like *clasp* [Gebser *et al.*, 2012b] contain several advanced features. One such feature is projection: given a set of atoms (or relations), the solver will output answer sets where all other atoms are projected away, and will also guarantee that there are no repetitions (even if multiple answer sets with the same assignment on the projected atoms exist), while still maintaining efficiency. This can be used to enumerate candidate world views by projecting away all relations in our encoding, except for $g(\cdot)$ and $v_{check_1}(\cdot)$. When enumerating all projected answer sets in this way, our encoding yields all guesses together with their candidate world views (when grouped by $g(\cdot)$).

4 The *selp* System

We implemented the reduction in the previous section as part of the single shot ELP solving toolbox *selp*, available at `dbai.tuwien.ac.at/proj/selp`. 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 *python* and depend on the parser generator *LARK*[3], the rule decomposition tool *lpopt* [Bichler *et al.*, 2016a], the tree decomposition tool *htd_main* [Abseher *et al.*, 2017], and the ASP grounder *gringo* [Gebser *et al.*, 2011].

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

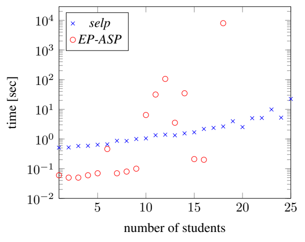
Toolbox. We briefly present the main building blocks of *selp*.

- ***easpGrounder.py*** takes as input a *non-ground EASP-not* program and outputs its equivalent ground form by rewriting it into an ASP program that the *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, *easpGrounder.py* supports arithmetics and the *sorts* format [Kahl *et al.*, 2015] as input.

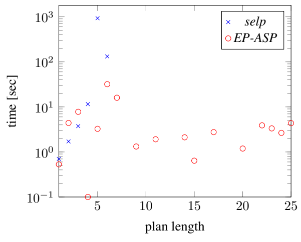
- ***easp2asp.py*** is *selp*'s key component. It takes a ground *EASP-not* program, and performs the reduction given in Section 3.1, also adhering to the practical considerations presented in Section 3.2, and finally outputs the resulting non-ground logic program in the syntax of *gringo*. Optionally,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

- ***groupWorldViews.py*** takes as input *clasp*'s output in JSON format, groups the answer sets into candidate world

[3]Available here: `https://github.com/erezsh/lark`



(a) Scholarship Eligibility



(b) Yale Shooting

Figure 2: Benchmark results. Missing points indicate timeouts.

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 -sELP | 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[4]. 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, *htd_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[5].

- *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.

[4]Note that to have a fair comparison we disabled the subset-maximality check on the guess that *EP-ASP* performs by default.

[5]Benchmark archive: `dbai.tuwien.ac.at/proj/selp`

- *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 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_Π of the ground ELP Π 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 Φ is known to be Σ_P^4-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

[Abseher *et al.*, 2017] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A free, open-source framework for (customized) tree decompositions and beyond. In *Proc. CPAIOR*, pages 376–386, 2017.

[Alviano *et al.*, 2013] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In *Proc. LPNMR*, pages 54–66, 2013.

[Bichler *et al.*, 2016a] Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A rule optimization tool for answer set programming. In *Proc. LOPSTR*, pages 114–130, 2016.

[Bichler *et al.*, 2016b] Manuel Bichler, Michael Morak, and Stefan Woltran. The power of non-ground rules in answer set programming. *TPLP*, 16(5-6):552–569, 2016.

[Bodlaender, 1993] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993.

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[del Cerro *et al.*, 2015] Luis Fariñas del Cerro, Andreas Herzig, and Ezgi Iraz Su. Epistemic equilibrium logic. In *Proc. IJCAI*, pages 2964–2970, 2015.

[Eiter and Gottlob, 1995] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995.

[Eiter *et al.*, 2007] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007.

[Faber *et al.*, 2011] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.

[Ferraris *et al.*, 2011] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artif. Intell.*, 175(1):236–263, 2011.

[Gebser *et al.*, 2011] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In *Proc. LPNMR*, pages 345–351, 2011.

[Gebser *et al.*, 2012a] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Morgan & Claypool, 2012.

[Gebser *et al.*, 2012b] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

[Gebser *et al.*, 2014] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *clingo* = ASP + control: Preliminary report. In *ICLP Tech.Comm.*, 2014.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP/SLP*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[Gelfond, 1991] Michael Gelfond. Strong introspection. In *Proc. AAAI, Volume 1*, pages 386–391, 1991.

[Gelfond, 1994] Michael Gelfond. Logic programming and reasoning with incomplete information. *Ann. Math. Artif. Intell.*, 12(1-2):89–116, 1994.

[Gelfond, 2011] Michael Gelfond. New semantics for epistemic specifications. In *Proc. LPNMR*, pages 260–265, 2011.

[Kahl *et al.*, 2015] Patrick Thor Kahl, Richard Watson, Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. The language of epistemic specifications (refined) including a prototype solver. *J. Log. Comput.*, 25, 2015.

[Kahl, 2014] Patrick Thor Kahl. *Refining the Semantics for Epistemic Logic Programs*. PhD thesis, Texas Tech University, Texas, USA, 2014.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Ann. Math. Artif. Intell.*, 25(3-4):369–389, 1999.

[Pearce, 2006] David Pearce. Equilibrium logic. *Ann. Math. Artif. Intell.*, 47(1-2):3–41, 2006.

[Pelov *et al.*, 2007] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *TPLP*, 7(3):301–353, 2007.

[Pulina, 2016] Luca Pulina. The ninth QBF solvers evaluation - preliminary report. In *Proc. QBF*, 2016.

[Shen and Eiter, 2016] Yi-Dong Shen and Thomas Eiter. Evaluating epistemic negation in answer set programming. *Artif. Intell.*, 237:115–135, 2016.

[Shen *et al.*, 2014] Yi-Dong Shen, Kewen Wang, Thomas Eiter, Michael Fink, Christoph Redl, Thomas Krennwallner, and Jun Deng. FLP answer set semantics without circular justifications for general logic programs. *Artif. Intell.*, 213:1–41, 2014.

[Son *et al.*, 2017] Tran Cao Son, Tiep Le, Patrick Thor Kahl, and Anthony P. Leclerc. On computing world views of epistemic logic programs. In *Proc. IJCAI*, pages 1269–1275, 2017.

[Truszczynski, 2011] Miroslaw Truszczynski. Revisiting epistemic specifications. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, 2011.