Evaluating Epistemic Logic Programs via Answer Set Programming with Quantifiers

Wolfgang Faber, Michael Morak
University of Klagenfurt, Austria
wolfgang.faber@aau.at, michael.morak@aau.at

Abstract

In this paper we introduce a simple way to evaluate epistemic logic programs by means of answer set programming with quantifiers, a recently proposed extension of answer set programming. The method can easily be adapted for most of the many semantics that were proposed for epistemic logic programs. We evaluate the proposed transformation on existing benchmarks using a recently proposed solver for answer set programming with quantifiers, which relies on QBF solvers.

1 Introduction

Answer Set Programming (ASP) is a generic, fully declarative logic programming language that allows users to encode problems such that the resulting output of the program (called *answer sets*) directly corresponds to solutions of the original problem (Gelfond and Lifschitz 1988, 1991; Brewka, Eiter, and Truszczynski 2011; Schaub and Woltran 2018).

Epistemic Logic Programs (ELPs) are an extension of ASP with epistemic operators. Originally introduced as the two modal operators $\mathbf{K}$ ("known" or "provably true") and $\mathbf{M}$ ("possible" or "not provably false") by Gelfond (Gelfond 1991, 1994), epistemic extensions of ASP have received renewed interest (c.f. e.g. (Gelfond 2011; Truszczynski 2011; Kahl 2014; del Cerro, Herzig, and Su 2015; Shen and Eiter 2016; Son et al. 2017; Kahl and Leclerc 2018; Faber, Morak, and Woltran 2019; Morak 2019)), with refinements of the semantics and proposals of new language features. Further, the development of efficient solving systems is underway with several efficient systems now available (Kahl et al. 2015; Son et al. 2017; Bichler, Morak, and Woltran 2020).

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

$$\mathit{innocent}(X) \leftarrow \neg\mathbf{K}\,\mathit{guilty}(X),$$

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

Recently, another language extension for ASP has been proposed, named ASP with Quantifiers, or ASP(Q) (Amendola, Ricca, and Truszczynski 2019), which introduces an explicit way to express quantifiers and quantifier alternations in ASP, reminiscent of Quantified Boolean Formulas (QBFs), but also different in spirit. It has the form

$$\Box_1 P_1 \Box_2 P_2 \cdots \Box_n P_n : C,$$

where the P_i are classical ASP programs, $\Box_i$ are quantifiers, and C expresses a set of constraints in ASP.

Example 2. *The intuitive reading of the ASP(Q) program*

$$\exists P_1 \forall P_2 : C$$

is that there exists an answer set A_1 of ASP program P_1 such that for each answer set A_2 of the ASP program P_2 combined with A_1 as facts, A_2 satisfies the constraints C.

Since evaluating ASP(Q) programs is PSPACE-complete in general (Amendola, Ricca, and Truszczynski 2019), this formalism forms an interesting target for rewriting ELPs. In this paper, we propose such a rewriting, where, given an ELP Π, we create an ASP(Q) program that is consistent if and only if Π has a world view. This happens by using the ASP(Q) quantifiers to directly encode the semantics of world views of ELPs, and, in turn, the existence the relevant stable models inside of that world view. We then experimentally verify that this encoding, together with a QBF solver-based ASP(Q) solver, indeed performs well, compared to current ELP solvers.

Contributions. The results and contributions presented in this paper can be summarized as follows.

- We specify a rewriting from ELPs to ASP(Q) programs in such a way that it preserves consistency. This rewriting also allows us to extract information about the world views of the original ELPs by evaluating the outermost quantifier block of the obtained ASP(Q) program. We show that our rewriting is flexible in the sense that it is applicable to all known semantics of ELPs, as long as they are based on the notion of an epistemic reduct.
- We implement a rewriting tool that performs our rewriting on real-world ELP instances under the well-known G94 semantics for ELPs (Gelfond 1991, 1994).
- We compare the performance of evaluating ELPs via our rewriting tool and state-of-the-art ASP(Q) solvers versus several existing ELP solvers. We observe that, indeed, our ASP(Q) rewriting approach shows competitive performance.

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Related Work. Most ELP solvers build upon an underlying ASP solver that is called multiple times in a procedural, sequential manner (Leclerc and Kahl 2018). The *selp* system (Bichler, Morak, and Woltran 2020) follows a similar approach to the one proposed in this paper, since it tries to rewrite an ELP into a non-ground ASP program with fixed arity in order to then use a single call to an ASP solver to establish the consistency of the input ELP. In this work, we try to follow a similar approach by rewriting ELPs to the language of ASP(Q). This is due to the fact that ASP(Q) allows one to easily express the quantification that is needed to write an intuitive encoding of the ELP semantics. However, other target languages that follow a similar idea would be available, including the stable-unstable semantics (Bogaerts, Janhunen, and Tasharrofi 2016), for which a solver implementation has recently been proposed (Janhunen 2022), but also the language of Quantified ASP (Fandinno et al. 2021), which follows a similar approach to ASP(Q), but does not quantify over answer sets but over atoms. We found the language of ASP(Q) to be very intuitive to use in practice, as well as having several solver implementations available, and hence chose this as our target language in this paper.

Structure. The remainder of the paper is structured as follows. In Section 2, we provide an overview of ELPs and ASP(Q). In Section 3, we present our rewriting of ELPs to ASP(Q). We then perform an experimental evaluation in Section 4. We conclude with a summary in Section 5.

2 Preliminaries

Answer Set Programming (ASP). We only give a very brief overview of the core language. For more information, we refer to standard literature (Brewka, Eiter, and Truszczynski 2011; Gebser et al. 2012; Lifschitz 2019), and, in our case, the ASP-Core-2 input language format (Calimeri et al. 2020).

Briefly, ASP programs consist of sets of *rules* of the form

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

where $n \geq m \geq l$ and for all $1 \leq i \leq n$, each a_i is an *atom* of the form $p(t_1, \ldots, t_n)$, where p is a predicate name, and $t_1, \ldots, t_n$ are terms, that is, either variables or constants. A *literal* ℓ is either an atom a or a negated atom $\neg a$. By $\overline{\ell}$ we denote the converse of literal ℓ, that is, if $\ell = a$, then $\overline{\ell} = \neg a$, and vice versa. The domain of constants in an ASP program P is given implicitly by the set of all constants that appear in it. Generally, before evaluating an ASP program, variables are removed by a process called *grounding*, that is, for every rule, each variable is replaced by all possible combination of constants, and appropriate ground copies of the rule are added to the resulting program $ground(P)$. In practice, several optimizations have been implemented in state-of-the-art grounders that try to minimize the size of the grounding while preserving equivalence. Since no more variables are present after grounding, *ground* ASP program can be defined simply as set of rules of form (1), where for all $1 \leq i \leq n$, each a_i is *ground* or *propositional atom*. A rule 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 (ground) rule is called a *fact* if it has the form $a \leftarrow \top$, for some atom a. It is called a *constraint* if it has the form $\bot \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$, with $n \geq m \geq 1$. The semantics of non-ground ASP programs are defined in terms of the semantics of their groundings. We will focus on ground ASP programs in the remainder of this paper.

The semantics of a (ground) ASP program P is as follows (Gelfond and Lifschitz 1988, 1991): An *interpretation* I (i.e. a set of ground atoms appearing in P) is called a *model* of P iff it satisfies all the rules in P in the sense of classical logic, that is, for each $r \in P$, $B^+(r) \subseteq M$ together with $B^-(r) \cap M = \emptyset$ implies that $H(r) \cap M \neq \emptyset$. We denote the set of models of r by $mods(r)$. The models of a program P are then given by $mods(P) = \bigcap_{r \in P} mods(r)$.

The reduct P^I of a program P with respect to an interpretation I is the program $P^I = \{r^I \mid r \in P, B^-(r) \cap I = \emptyset\}$, where $r^I = H(r) \leftarrow B^+(r)$; that is, P^I is defined as the set of rules obtained from P where all negated atoms on the right-hand side of the rules are evaluated over I and replaced by $\top$ or $\bot$ accordingly.

A model M is called an *answer set* of program P if $M \in mods(P)$ and there is no $M' \subseteq M$ such that $M' \in mods(P^M)$. The set of answer sets of a program P is denoted $AS(P)$. The *consistency problem* of ASP (decide whether for a given program P, $AS(P) \neq \emptyset$) is Σ_P^2-complete[1] (Eiter and Gottlob 1995).

In this paper, we will make limited use of so-called choice rules as defined in (Calimeri et al. 2020). For an atom a, such a *(limited) choice rule* r is denoted $\{a\} \leftarrow \top$. The reduct r^I of r is $a \leftarrow \top$ if $a \in I$, and otherwise $a \leftarrow \bot$. That is, choice rule $\{a\}$ induces two answer sets: one where a is true, and one where a is false.

Answer Set Programming with Quantifiers (ASP(Q)). An extension of ASP, referred to as ASP(Q), has been proposed (Amendola, Ricca, and Truszczynski 2019), providing a formalism reminiscent of Quantified Boolean Formulas, but based on ASP. An ASP(Q) program is of the form

$$\Box_1 P_1 \Box_2 P_2 \cdots \Box_n P_n : C,$$

where, for each $i \in \{1, \ldots, n\}$, $\Box_i \in \{\exists, \forall\}$, P_i is an ASP program, and C is a set of constraints (technically, it can be a stratified normal ASP program and is intended by the ASP(Q) authors as a "check" at the very end). $\exists$ and $\forall$ are called *existential* and *universal answer set quantifiers*, respectively.

As a brief example, the intuitive reading of an ASP(Q) program $\exists P_1 \forall P_2 : C$ is that there exists an answer set A_1 of P_1 such that for each answer set A_2 of $P_2 \cup A_1$ it holds that $C \cup A_2$ is consistent (i.e. has an answer set).

Let us be more precise about the program $P \cup A$, that is, a program P being extended by an answer set A (or rather by an interpretation A): For an interpretation I, let $f_P(I)$ be the ASP program that contains all atoms in I as facts and all

[1] Note that this complexity holds for ground programs. In the non-ground case, the complexity of deciding ASP consistency is $\text{NExpTime}^{\text{NP}}$-complete in general (Dantsin et al. 2001).

atoms a appearing in P but not in I as constraints (i.e. as a rule $\bot \leftarrow a$). Furthermore, for a program P and an interpretation I, let $f_P(\Pi, I)$ be the ASP(Q) program obtained from an ASP(Q) program Π by replacing the first program P_1 in Π with $P_1 \cup f_P(I)$. *Coherence* of an ASP(Q) program is then defined inductively:

- $\exists P : C$ is coherent if there exists an answer set M of P such that $C \cup f_P(M)$ has at least one answer set.
- $\forall P : C$ is coherent if for all answer sets M of P it holds that $C \cup f_P(M)$ has at least one answer set.
- $\exists P \Pi$ is coherent if there exists an answer set M of P such that $f_P(\Pi, M)$ is coherent.
- $\forall P \Pi$ is coherent if for all answer sets M of P it holds that $f_P(\Pi, M)$ is coherent.

In addition, for an existential ASP(Q) program Π (one that starts with $\exists$), the answer sets of the first ASP program P_1 are referred to as *quantified answer sets*.

In general, deciding coherence for an ASP(Q) program is known to be PSPACE-complete (Amendola, Ricca, and Truszczynski 2019, Theorem 2), and on the n-th level of the polynomial hierarchy for programs with n quantifier alternations (Amendola, Ricca, and Truszczynski 2019, Theorem 3).

Epistemic Logic Programming (ELP). An *epistemic* or *subjective literal* is a formula $\mathbf{K}\,\ell$ or $\mathbf{M}\,\ell$, where ℓ is a literal and $\mathbf{K}$ and $\mathbf{M}$ are the epistemic operators of *certainty* (“known” or “provably true”) and *possibility* (“maybe,” “possible,” or “not provably false”). An *epistemic logic program (ELP)* 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 $k \geqslant 0$, $m \geqslant 0$, $n \geqslant j \geqslant 0$, each a_i is an atom, each ℓ_i is a literal, and each ξ_i is an epistemic literal. Such rules are also called *ELP rules*. Similarly to ASP, we consider an ELP *ground*, if no variables appear in it, and treat programs with variables as an abbreviation of the ground program, where each variable is replaced with every possible constant from the program. In this paper, we deal with ground programs.

Let $H(r) = \{a_1, \ldots, a_k\}$ denote the head elements of an ELP rule, 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. The set of epistemic literals occurring in an ELP Π is denoted $elit(\Pi)$.

The semantics of ELPs are given via so-called world views. An *epistemic interpretation* $\mathcal{I} = \{I_1, \ldots, I_n\}$ of an ELP Π is a set of ASP interpretations w.r.t. the atoms appearing in Π, with $n \geq 0$. We say that $\mathbf{K}\,\ell$ is true in $\mathcal{I}$, if and only if ℓ is true in all interpretations in $\mathcal{I}$, and $\mathbf{M}\,\ell$ is true in $\mathcal{I}$, if and only if ℓ is true in at least one interpretation in $\mathcal{I}$. Let $\mathcal{I}$ be an epistemic interpretation; then $\Pi_S^{\mathcal{I}}$ denotes the *S-epistemic reduct* of Π w.r.t. $\mathcal{I}$, which is derived from Π and $\mathcal{I}$ according to some reduct-based semantics S.

Definition 3. *An epistemic interpretation $\mathcal{I}$ is called an S-*world view *of an ELP Π iff $\mathcal{I} = AS(\Pi_S^{\mathcal{I}})$ and $\mathcal{I} \neq \emptyset$.*

Various semantics S have been defined over the years and most of them can be characterized by reducts $\Pi_S^{\mathcal{I}}$, either over sets of interpretations (as above) or over the set of $elit(\Pi)$ that are satisfied by $\mathcal{I}$ (Kahl 2014; del Cerro, Herzig, and Su 2015; Shen and Eiter 2016; Son et al. 2017; Kahl and Leclerc 2018; Faber, Morak, and Woltran 2019; Morak 2019; Cabalar, Fandinno, and del Cerro 2020). A concise summary can be found in (Leclerc and Kahl 2018). The formal results presented in this paper generally hold for all reduct-based semantics, that is, for semantics that constructs $\Pi_S^{\mathcal{I}}$ by eliminating the epistemic literals from Π by evaluating them over the given epistemic interpretation $\mathcal{I}$. In particular, we will use the G94 semantics (Gelfond 1991, 1994) as a running example and as the basis for our practical evaluation. The G94 semantics are defined as follows: $\Pi_{G94}^{\mathcal{I}}$ denotes the *G94-epistemic reduct* of Π w.r.t. $\mathcal{I}$, which is obtained from Π by replacing all epistemic literals in Π with either $\top$ or $\bot$, depending on whether they are true or false in $\mathcal{I}$, respectively. When the semantics used are clear or not relevant, we will often refer to the S-epistemic reduct simply as the *epistemic reduct*.

The main reasoning task for ELPs is checking whether they are *consistent*, that is, whether they have a world view. This problem is also referred to as *world view existence problem* and is generally known to be Σ_3^P-complete in the ground case for most known reduct-based semantics (Shen and Eiter 2016), and, in particular, the G94 semantics.

3 Transformation

Based on the semantics described in Section 2, evaluating (ground) ELPs can be seen as a three-step procedure in order to establish world view existence. Given an ELP Π:

1. Guess an epistemic interpretation $\mathcal{I}$.
2. For the given reduct-based semantics S under consideration, construct the epistemic reduct $\Pi_S^{\mathcal{I}}$.
3. Finally, evaluate the ASP program $\Pi_S^{\mathcal{I}}$ and check that $\mathcal{I} = AS(\Pi_S^{\mathcal{I}})$, that is, that the answer sets of the epistemic reduct coincide with the originally guessed interpretation $\mathcal{I}$.

Guessing an entire epistemic interpretation is, however, tricky in practice, since it can be exponentially larger than the program Π. However, the procedure above can be simplified, since it has been shown that a world view is uniquely determined by the set of epistemic literals that it entails (Morak 2019). Hence, if checking world view existence is the goal (as opposed to actually constructing a world view), in Step 1 it actually suffices to guess, for each epistemic literal in Π, whether it will ultimately be entailed by the world view. This is also enough information to construct the relevant epistemic reduct in Step 2. Let Φ denote this guess, and let Π_S^{Φ} denote the reduct obtained from guess Φ. Now, in Step 3, it needs to be checked whether the truth assignment to the epistemic literals from Step 1 is consistent with $AS(\Pi_S^{\Phi})$.

Following the characterization of world views presented above, in this section, we propose a rewriting for checking for world view existence using the language of Answer Set Programming with Quantifiers—ASP(Q). The approach is similar to the one used in the ELP solver *selp* (Bichler, Morak, and Woltran 2020), which encodes ground ELPs as

non-ground ASP programs. The general structure is as follows: Given an ELP Π, our ASP(Q) rewriting first "guesses" a truth assignment Φ for the epistemic literals in Π, using an existentially quantified ASP program. In order to establish existence of a world view w.r.t. this guess Φ, three conditions must be verified:

1. $AS(\Pi_S^\Phi) \neq \emptyset$.
2. For each epistemic literal $\mathbf{M}\,\ell$ (resp. $\mathbf{K}\,\ell$) that is guessed as "true" (resp. "false") in Φ, we need to verify that ℓ is true (resp. false) in at least one answer sets of Π_S^Φ.
3. For each epistemic literal $\mathbf{K}\,\ell$ (resp. $\mathbf{M}\,\ell$) that is guessed as "true" (resp. "false") in Φ, we need to verify that ℓ is true (resp. false) in all answer sets of Π_S^Φ.

For Condition 1, we use an existentially quantified ASP program to establish that the epistemic reduct has at least one answer set.

For each epistemic literal of Condition 2, an existentially quantified ASP program is used to check that there is indeed an answer set of the epistemic reduct witnessing the guessed truth value of the epistemic literal.

For Condition 3, a single universally quantified ASP program, together with the final set of constraints, is then used to check that, indeed, ℓ is true in every answer set of the epistemic reduct w.r.t. the guess.

3.1 Reducing ELPs to ASP(Q) Programs

In this section, we present the formal rewriting of an ELP into ASP(Q) according to semantics S, according to the outline presented above. Hence, given an ELP Π, we construct an ASP(Q) program Π', which is structured as follows, where $elit(\Pi) = \{\xi_1, \ldots, \xi_n\}$:

$$\Pi' = \exists P_\Phi \exists P_\exists \exists P_1 \ldots \exists P_n \forall P_\forall : C.$$

In this construction, sub-program P_Φ will guess a truth assignment for the epistemic literals present in program Π. Then, sub-program $P_\exists$ will check Condition 1 stated above, sub-programs $P_1, \ldots P_n$ together will collectively check Condition 2, and finally sub-program $P_\forall$ will check Condition 3. Let us now give the construction of these sub-programs one by one. To this end, let $\widehat{\xi}$ denote a fresh atom, not occurring in Π, representing the epistemic literal ξ. To illustrate how our reduction works, we will use a simple running example under the G94 semantics, introduced below.

Example 4 (Running Example)**.** *The following ELP* Π_{ex}*,*

$$a \leftarrow \mathbf{K}\,\neg b$$
$$b \leftarrow \mathbf{K}\,\neg a,$$

interpreted under the G94 semantics, has two world views: $\{\{a\}\}$ *and* $\{\{b\}\}$*.*

Sub-Program P_Φ. Here, we simply need to guess, for every epistemic literal present in ELP Π, whether it is supposed to be true or false in the resulting world view (if such a world view exists for that guess). Hence, for each $\xi \in elit(\Pi)$, P_Φ contains exactly one rule of the following form:

$$\{\widehat{\xi}\} \leftarrow \top$$

This completes the construction of P_Φ. By using a choice rule for each epistemic literal, the resulting answer sets of P_Φ each represent exactly one truth assignment to the epistemic literals in Π.

Example 5. *Continuing on from Example 4,* P_Φ *would consist of the following two rules:*

$$\{kna\} \leftarrow \top$$
$$\{knb\} \leftarrow \top,$$

where $kna = \widehat{\mathbf{K}\,\neg a}$ *is an atom representing the epistemic literal* $\mathbf{K}\,\neg a$*, and similarly for* knb *represents* $\mathbf{K}\,\neg b$*.* P_Φ *has four answer sets:* $\emptyset$*,* $\{kna\}$*,* $\{knb\}$*, and* $\{kna, knb\}$*, representing four possible world views w.r.t. the two epistemic literals in* Π_{ex}*.*

Sub-Programs $P_\exists$. In order to construct the sub-program in this paragraph, we first need to introduce the notion of the encoded epistemic reduct. Given a reduct-based semantics S for ELPs, we recall that Π_S^Φ denotes the S-epistemic reduct w.r.t. a guess Φ on the epistemic literals in Π. In our encoding, however, we cannot directly construct this reduct, since we don't know Φ (since it is constructed within the rewritten program itself via sub-program P_Φ). Hence, let Π_S^* denote the *encoded S-epistemic reduct* of Π, that is, an ASP program, such that the following equivalence holds, for all subsets $\Phi \subseteq elit(\Pi)$:

$$AS(\Pi_S^* \cup \widehat{\Phi}) = AS(\Pi_S^\Phi),$$

where $\widehat{\Phi} = \{\widehat{\xi} \mid \xi \in \Phi\}$. We can show that such an encoded reduct always exists: S-epistemic reducts are obtained by replacing the epistemic literals in Π with some fixed set of ASP literals, depending on S and the guess Φ. But this can always be encoded using ASP: Take rule r containing epistemic literal ξ and duplicate it. In the one copy, replace ξ with $\widehat{\xi} \cup \mathcal{L}^+$ (where $\mathcal{L}^+$ is the set of ASP literals according to semantics S for the case where $\xi \in \Phi$). In the other copy, replace ξ with $\neg\widehat{\xi} \cup \mathcal{L}^-$ (where $\mathcal{L}^-$ is the set of ASP literals according to semantics S for the case where $\xi \notin \Phi$). The resulting program Π_S^* clearly is an encoded S-epistemic reduct according to the definition above, since for each set of atoms $\widehat{\Phi}$ added to Π_S^*, for each rule r of Π, only one of the two copies will ever be "triggerable", since the body of the other copy is trivially satisfied by construction.

Now, for an ASP program P, let $[P]_o$ be the same ASP program, where each atom a not in $\{\widehat{\xi} \mid \xi \in elit(\Pi)\}$ is renamed to a_o, where o is an arbitrary string. We will use this to create several independent copies of program P that do not interact with each other because they share no atoms, except the atoms representing the guess Φ.

Now, to finalize the construction of the sub-programs in this paragraph, let

- $P_\exists = [\Pi_S^*]_\exists$.

This completes the construction of $P_\exists$. The intention of the existentially quantified program $P_\exists$ is to ensure Condition 1 of the three conditions stated in the introduction to this section. It ensures that there is at least one answer set of the epistemic reduct of Π w.r.t. the guess Φ.

Example 6. *Continuing on from Example 5, $P_\exists$ would consist of the following rules under the G94 semantics:*

$$a_\exists \leftarrow knb.$$
$$b_\exists \leftarrow kna.$$

Note that the G94-epistemic reduct replaces an epistemic literal with $\top$ if true and with $\bot$ if false. This behaviour is represented in the encoded G94-epistemic reduct by simply replacing each epistemic literal ξ with the atom representing the truth value of ξ in guess Φ, as determined by the sub-program P_Φ.

The existentially quantified program $P_\exists$ above ensures that, whatever guess Φ is made, the resulting epistemic reduct has at least one answer set, and hence, ensures Condition 1.

Sub-Programs $P_1, \ldots, P_n$. In order to ensure Condition 2, we need separate existentially quantified programs $P_1, \ldots, P_n$ to separately establish the existence of an appropriate answer set of the epistemic reduct, for each epistemic literal, when the truth value in Φ is the one in Condition 2. With the already introduced formal notions in place, this is simply done by creating several copies of the encoded epistemic reduct, each enhanced with specific constraints establish the property in Condition 2. To this end, let

- $P_i = [\Pi^*_S]_i \cup \{c_i\}$, for each $0 < i \leq n$,

where c_i is a single constraint for epistemic literal ξ_i as follows: In case $\xi_i = \mathbf{K}\,\ell$, then c_i is the constraint

$$\bot \leftarrow \neg\widehat{\xi_i}, \ell,$$

whereas when $\xi_i = \mathbf{M}\,\ell$, then c_i is the constraint

$$\bot \leftarrow \widehat{\xi_i}, \overline{\ell}.$$

Indeed, whenever $\mathbf{K}\,\ell$ is guessed as false in Φ, this ensures that there is an answer set of the epistemic reduct where ℓ is false, and, similarly, whenever $\mathbf{M}\,\ell$ is guessed as true in Φ, this ensures that there is an answer set of the epistemic reduct where ℓ is true, establishing Condition 2.

Example 7. *Continuing on from Example 6, P_1 (with $\xi_1 = \mathbf{K}\,\neg b$) would consist of the following rules under the G94 semantics:*

$$a_1 \leftarrow knb.$$
$$b_1 \leftarrow kna.$$
$$\bot \leftarrow \neg knb, \neg b_1.$$

The existentially quantified program P_1 above ensures that, whenever epistemic literal $\xi_1 = \mathbf{K}\,\neg b$ is guessed as false in P_Φ, the resulting epistemic reduct has at least one answer set where b is true, establishing Condition 2 for ξ_1.

Sub-Program $P_\forall$ and Constraints C. We finally need to establish Condition 3. This is done via the universally quantified ASP program $P_\forall$ which, together with the constraints C, will verify that Condition 3 holds.

While for Condition 2 we needed separate, independent sub-programs to establish answer set existence within the world view for each relevant epistemic literal, Condition 3 imposes a universal condition that needs to hold in all answer sets. The idea of $P_\forall$ is therefore to quantify over all answer sets of the relevant epistemic reduct and ensure consistency with the epistemic literals in guess Φ. Since Condition 3 must hold in all answer sets, one universally quantified program is enough to ensure this for all relevant epistemic literals in guess Φ.

To this end, program let program $P_\forall$ be constructed as follows:

- $P_\forall = [\Pi^*_S]_\forall$,

and let the set of constraints C contain the following:

- $\bot \leftarrow \widehat{\xi}, \overline{\ell}$, for each epistemic literal $\xi = \mathbf{K}\,\ell$ in $elit(\Pi)$; and
- $\bot \leftarrow \neg\widehat{\xi}, \ell$, for each epistemic literal $\xi = \mathbf{M}\,\ell$ in $elit(\Pi)$.

This completes the construction of $P_\forall$ and C, and hence of our reduction. Since $P_\forall$ is a universally quantified ASP program within our ASP(Q) rewriting Π', every answer set of $P_\forall$ must satisfy all the constraints imposed in C. Since the answer sets of $P_\forall$ are precisely the answer sets of the epistemic reduct Π^Φ_S, this indeed ensures precisely Condition 3: in case where $\xi = \mathbf{K}\,\ell$ is guessed as true in Φ, ℓ must be true in all answer sets of the epistemic reduct, and in case where $\xi = \mathbf{M}\,\ell$ is guessed as false in Φ, ℓ must be false in all answer sets of the epistemic reduct.

Example 8. *Continuing on from Example 7, $P_\forall$ would consist of the following rules under the G94 semantics:*

$$a_\forall \leftarrow knb.$$
$$b_\forall \leftarrow kna.$$

The set C would consist of the following two constraints:

$$\bot \leftarrow knb, b_\forall.$$
$$\bot \leftarrow kna, a_\forall.$$

$P_\forall$ simply represents a copy of the epistemic reduct w.r.t. guess Φ. The constraints then ensure that, (1) whenever P_Φ guesses epistemic literal $\mathbf{K}\,\neg b$ to hold within the world view, it cannot be the case that in any answer set of the epistemic reduct b holds; and (2) whenever P_Φ guesses epistemic literal $\mathbf{K}\,\neg a$ to hold within the world view, it cannot be the case that in any answer set of the epistemic reduct a holds. This establishes precisely Condition 3.

Uniting the code of Examples 4, 5, 6, 7, and this one, obtaining program ASP(Q) program Π', it is not difficult to check that indeed Conditions 1–3 are satisfied under the G94 semantics for the original ELP Π, whenever Π' is coherent.

3.2 Correctness

With the above reduction in place, we can first observe that the reduction is polynomial in size: it repeats the epistemic reduct construction (which itself is only linear in size) n times, where n is the number of epistemic literals in the ELP. We now need to show that it is indeed correct:

Theorem 9. *Let Π be an ELP, and S be a reduct-based semantics for interpreting ELPs. Let Π' be the ASP(Q) program obtained by applying the transformation given in Section 3.1 to Π under S. Then Π has at least one world view if and only if Π' is coherent.*

Proof (Sketch). We show both directions separately. To this end, let $\Pi' = \exists P_\Phi \exists P_\exists \exists P_1 \ldots \exists P_n \forall P_\forall : C$, as in Section 3.1, and let $elit(\Pi) = \{\xi_1, \ldots, \xi_n\}$.

$\Rightarrow$**:** Assume that Π has at least one world view. We will show that Π' is coherent. Let $\mathcal{I}$ be a world view of Π under semantics S. Let $\Phi \subseteq elit(\Pi)$ be the set of epistemic literals true in $\mathcal{I}$. Clearly, there is a model of P_Φ reflecting precisely this "guess" Φ. Since $\mathcal{I}$ is a world view of Π, we have, by Definition 3 (cf. Morak (2019), Theorem 11) that $\mathcal{I} = AS(\Pi_S^\Phi)$. But then, clearly, Conditions 1–3 must be satisfied in the epistemic reduct Π_S^Φ. But these three conditions are encoded, by construction, into $P_\exists$, P_i ($0 < i \leq n$), and $P_\forall : C$, respectively. Hence, by virtue of $\mathcal{I}$ being a world view, we have that (a) Condition 1 holds for Π_S^Φ by assumption, and hence $P_\exists$ has at least one answer set; (b) P_i ($0 < i \leq n$) has at least one answer set that witnesses the truth assignment to epistemic literal ξ_i in Φ according to Condition 2, since that condition holds for Π_S^Φ by assumption; and (c) every answer set of $P_\forall$ satisfies all the constraints in C, since they encode precisely Condition 3, and again that condition holds for Π_S^Φ by assumption. Hence, we have that indeed there exists an answer set for P_Φ and for $P_\exists$, $P_1, \dots, P_n$, and for all answer sets of $P_\forall$, the constraints in C are satisfied. Hence Π' is coherent.

$\Leftarrow$**:** Assume that Π' is coherent. By this assumption, there exists an answer set for $P_\exists$ encoding a guess $\Phi \subseteq elit(\Pi)$. By assumption, there exists an answer set of $P_\exists$, and hence, by construction, there exists an answer set of Π_S^Φ, establishing Condition 1. Again by assumption, each program P_i ($0 < i \leq n$) has an answer set that, by construction, represents an answer set of Π_S^Φ that witnesses Condition 2 for one particular epistemic literal ξ_i, together establishing Condition 2 for Π_S^Φ. Finally, by assumption all answer sets of $P_\forall$ satisfy the constraints in C. But, by construction, these constraints can only be satisfied if Π_S^Φ satisfies Condition 3, since they precisely encode it. Hence, we have that Conditions 1–3 are all satisfied for Π_S^Φ, which is a sufficient condition for the existence of a world view of Π (Morak 2019, Theorem 11). This completes the proof. $\square$

We have established that our translation from ELPs to ASP(Q) programs faithfully mimics the semantics of the ELP program and can hence be used as a way to solve ELP programs by means of an ASP(Q) solver. In the next section, we implement this rewriting approach to see its performance compared to existing ELP solvers.

4 Experimental Evaluation

We tested the rewriting approach described in Section 3 for the G94 semantics, by benchmarking it against existing ELP solvers. We will refer to our rewriting tool as *elp2qasp*. To compare, we chose the state-of-the-art ELP solver *EP-ASP* (Son et al. 2017) and the *selp* solver (Bichler, Morak, and Woltran 2020) based on a rewriting to plain ASP. For our rewriting, we used two ASP(Q) solvers as backends: the *qasp* solver (Natale 2021), as well as the *q_asp* solver (Cuteri 2022). We partly re-use *selp*'s parsing implementation.

We use the same three test sets proposed in (Bichler, Morak, and Woltran 2020). For every test set, we measured the time it took to solve the consistency problem. For *selp*, the underlying ASP solver *clingo* (Gebser et al. 2019) was stopped after finding the first answer set. For *EP-ASP*, search was terminated after finding the first candidate world view[2]. For *qasp* and *q_asp*, the output of our ELP to ASP(Q) rewriting directly tells us whether the ELP is consistent or not, depending on whether the ASP(Q) program is consistent.

Experiments were run on an AMD EPYC 7601 system (2.2GHz base clock speed) with 500 GiB of memory. Each process was assigned a maximum of 14 GB of RAM, which was never exceeded by any of the solvers tested. A time limit of 900 seconds was used for each benchmark set. For *EP-ASP*, we made trivial modifications to the python code in order for it to run with *clingo* 5.4.1. For *selp* and *qasp*, the same version of *clingo* was used. For *selp*, in addition, we used the *htd* library, version 1.2.0, and *lpopt* 2.2. We used *qasp* 1.1.0 and *q_asp* 0.1.2 as the backend solvers for our ASP(Q) rewriting generated by *elp2qasp*. The time it took to convert input ELP programs into the specific input formats of the various tools we used (e.g. the input format for *selp* or *EP-ASP*) was not measured, since we did not want the input format conversion to influence the benchmark results. *EP-ASP* was called with the preprocessing option for brave and cautious consequences on, since it always ran faster this way. The time for *selp*, *qasp*, and *q_asp* is the sum of the time it took to run all required components to solve the relevant instance. For *selp*, *clingo* was always called with SAT preprocessing enabled, as is recommended by the *lpopt* tool.

4.1 Benchmark Instances

We used three types of benchmarks, two coming from the ELP literature and one from the QSAT domain[3]. This is the same benchmark set as used and published by the authors of the *selp* solver, which they used in the associated conference publication (Bichler, Morak, and Woltran 2020). We briefly describe the benchmark set below.

Scholarship Eligibility. 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. 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 from (Kahl et al. 2015).

Tree QBFs. The hardness proof for ELP consistency (Shen and Eiter 2016) relies on a reduction from the validity problem for restricted quantified boolean formulas with three quantifier blocks (i.e. 3-QBFs), which can be generalized to arbitrary 3-QBFs (Bichler, Morak, and Woltran 2020). In that publication, the reduction is applied to the 14 "Tree" instances of QBFEVAL'16 (Pulina 2016), available at http://www.qbflib.org/family_detail.php?idFamily=56.

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

[3]See supplementary material.

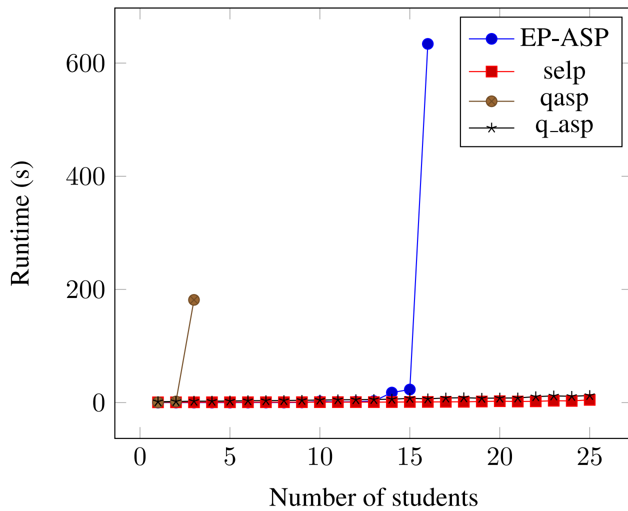

Figure 1: Scholarship Eligibility

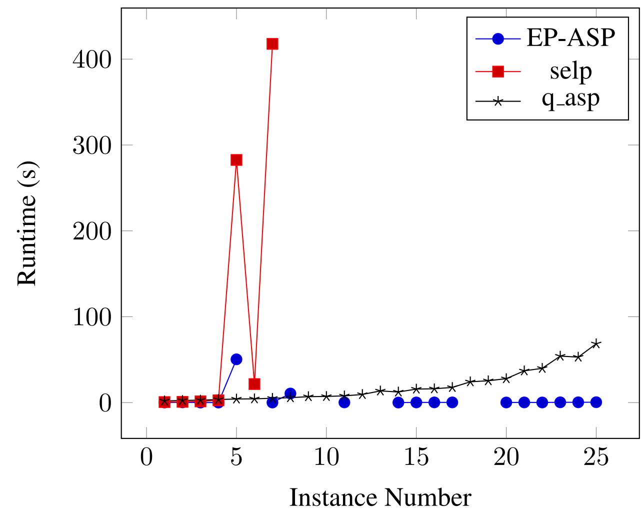

Figure 2: Yale Shooting Problem

4.2 Results

The results for the first two sets are shown in Figure 1 and Figure 2, respectively. For the Scholarship Eligibility Problem, we can observe, confirming the observations Bichler, Morak, and Woltran (2020), that *EP-ASP* can solve 16 instances, while *selp* is able to solve all instances, independent of time, within 5 seconds. Interestingly, the same holds true for our *elp2qasp* rewriting, when using *q_asp* as a backend. This combination can solve all instances within 13 seconds. Interestingly, the *qasp* backend is only able to solve three instances successfully within the time limit. The difference in performance between the two tools may be due to the fact that *q_asp* uses a QBF solver, while *qasp* delegates the solving work to the ASP solver *clingo* or *wasp* (Alviano et al. 2015). In all our benchmarks we use the latter option.

For the Yale Shooting Problem, we can see that both *EP-ASP* and *selp* are unable to solve all instances. Note that all instances of this problem are inconsistent, which sometimes allows *EP-ASP* to realize this fairly quickly. However, in seven cases, we dont get any answers from *EP-ASP* within the time limit. On the other hand, our *elp2qasp* approach with the *q_asp* backend is able to solve all instances of this problem within 70 seconds, with the solving time increasing moderately with the increase in the time horizon. For the *qasp* backend, we unfortunately encountered an issue in the implementation, which lead to an internal error message for all instances of the Yale Shooting Problem.

Finally, for the Tree QBF Problem, we re-confirmed the results of Bichler et al. (Bichler, Morak, and Woltran 2020). *selp* was able to solve 4 of the 14 instances within the time limit of 900 seconds. Both *EP-ASP* and *elp2qasp* with the *qasp* backend were unable to solve any instances at all. For the *q_asp* backend, this class of problems is unsolvable, since the resulting ASP(Q) rewriting contains rules that are not head-cycle free and are hence not treatable by *q_asp*. Since *selp* was the only solver able to successfully solve instances of this problem, we omit a dedicated figure.

These results confirm that *elp2qasp* is competitive for solving ELP programs, especially when paired with the *q_asp* solver for ASP(Q), which, in turn, is based on an internal QBF solver. On the other hand, the ASP(Q) solver *qasp* does not seem to match this success, but this may be an inherent limitation, since it internally relies on the ASP solver *clingo* or *wasp* to solve ASP(Q) instances, which may lead to an exponential number of internal ASP solver calls.

5 Conclusions

In this paper, we proposed a rewriting that transforms epistemic logic programs (ELPs) into programs for answer set programming with quantifiers (ASP(Q)). It does this by faithfully mimicking the semantics of ELPs and formulating them directly in ASP(Q), which is possible because of the explicit support for quantification that ASP(Q) provides.

We then implement our approach and, using state-of-the-art ASP(Q) solvers as a backend, test our rewriting approach against existing solvers for ELPs. We show that, for several problem domains, our rewriting approach offers competitive performance when compared to existing solvers, especially in case where the *q_asp* ASP(Q) solver (Cuteri 2022) is used, which internally uses a QBF solver to evaluate the given ASP(Q) program.

Future work includes further refining and optimizing the rewriting, as well as adapting our tool for the various other semantics that exist for evaluating ELPs (Kahl 2014; del Cerro, Herzig, and Su 2015; Shen and Eiter 2016; Son et al. 2017; Kahl and Leclerc 2018; Faber, Morak, and Woltran 2019; Morak 2019; Cabalar, Fandinno, and del Cerro 2020).

Another avenue of investigation is that our ASP(Q) program is capable of computing the guess Φ that gives rise to a world view, if one exists. This corresponds to a world view according to Morak (2019). However, several semantics employ a type of knowledge minimization (Shen and Eiter 2016; Cabalar, Fandinno, and del Cerro 2020) where we would need to extend our rewriting to capture this step.

References

Alviano, M.; Dodaro, C.; Leone, N.; and Ricca, F. 2015. Advances in WASP. In Calimeri, F.; Ianni, G.; and Truszczynski, M., eds., *Proc. LPNMR*, volume 9345 of *LNCS*, 40–54. Springer.

Amendola, G.; Ricca, F.; and Truszczynski, M. 2019. Beyond NP: Quantifying over Answer Sets. *Theory Pract. Log. Program.*, 19(5-6): 705–721.

Bichler, M.; Morak, M.; and Woltran, S. 2020. selp: A Single-Shot Epistemic Logic Program Solver. *Theory Pract. Log. Program.*, 20(4): 435–455.

Bogaerts, B.; Janhunen, T.; and Tasharrofi, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *Theory Pract. Log. Program.*, 16(5-6): 570–586.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM*, 54(12): 92–103.

Cabalar, P.; Fandinno, J.; and del Cerro, L. F. 2020. Autoepistemic answer set programming. *Artif. Intell.*, 289: 103382.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 Input Language Format. *Theory Pract. Log. Program.*, 20(2): 294–309.

Cuteri, B. 2022. Quantified ASP solver q_asp. https://github.com/bernardocuteri/q_asp. Accessed: 2023-04-04.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3): 374–425.

del Cerro, L. F.; Herzig, A.; and Su, E. I. 2015. Epistemic Equilibrium Logic. In *Proc. IJCAI*, 2964–2970.

Eiter, T.; and Gottlob, G. 1995. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Ann. Math. Artif. Intell.*, 15(3-4): 289–323.

Faber, W.; Morak, M.; and Woltran, S. 2019. Strong Equivalence for Epistemic Logic Programs Made Easy. In *Proc. AAAI*.

Fandinno, J.; Laferrière, F.; Romero, J.; Schaub, T.; and Son, T. C. 2021. Planning with Incomplete Information in Quantified Answer Set Programming. *Theory Pract. Log. Program.*, 21(5): 663–679.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *TPLP*, 19(1): 27–82.

Gelfond, M. 1991. Strong Introspection. In Dean, T. L.; and McKeown, K. R., eds., *Proc. AAAI*, 386–391. AAAI Press / The MIT Press.

Gelfond, M. 1994. Logic Programming and Reasoning with Incomplete Information. *Ann. Math. Artif. Intell.*, 12(1-2): 89–116.

Gelfond, M. 2011. New Semantics for Epistemic Specifications. In *Proc. LPNMR*, 260–265.

Gelfond, M.; and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In *Proc. ICLP/SLP*, 1070–1080.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Gener. Comput.*, 9(3/4): 365–386.

Janhunen, T. 2022. Implementing Stable-Unstable Semantics with ASPTOOLS and Clingo. In Cheney, J.; and Perri, S., eds., *Proc. PADL*, volume 13165 of *Lecture Notes in Computer Science*, 135–153. Springer.

Kahl, P. T. 2014. *Refining the Semantics for Epistemic Logic Programs*. Ph.D. thesis, Texas Tech University, Texas, USA.

Kahl, P. T.; and Leclerc, A. P. 2018. Epistemic Logic Programs with World View Constraints. In Palù, A. D.; Tarau, P.; Saeedloei, N.; and Fodor, P., eds., *Proc. ICLP*, volume 64 of *OASIcs*, 1:1–1:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Kahl, P. T.; Watson, R.; Balai, E.; Gelfond, M.; and Zhang, Y. 2015. The Language of Epistemic Specifications (Refined) Including a Prototype Solver. *J. Log. Comput.*, 30(4): 953–989.

Leclerc, A. P.; and Kahl, P. T. 2018. A survey of advances in epistemic logic program solvers. *CoRR*, abs/1809.07141. Also in Proc. of ASPOCP 2018.

Lifschitz, V. 2019. *Answer Set Programming*. Springer. ISBN 978-3-030-24657-0.

Morak, M. 2019. Epistemic Logic Programs: A Different World View. In *Proc. ICLP, Technical Communications*, volume 306 of *EPTCS*, 52–64.

Natale, A. 2021. Design and implementation of QASP Solver. https://zenodo.org/record/5425783. Accessed: 2023-04-04.

Pulina, L. 2016. The Ninth QBF Solvers Evaluation - Preliminary Report. In *Proc. QBF*, 1–13.

Schaub, T.; and Woltran, S. 2018. Special Issue on Answer Set Programming. *Künstliche Intell.*, 32(2-3): 101–103.

Shen, Y.; and Eiter, T. 2016. Evaluating epistemic negation in answer set programming. *Artif. Intell.*, 237: 115–135.

Son, T. C.; Le, T.; Kahl, P. T.; and Leclerc, A. P. 2017. On Computing World Views of Epistemic Logic Programs. In *Proc. IJCAI*, 1269–1275.

Truszczynski, M. 2011. Revisiting Epistemic Specifications. In Balduccini, M.; and Son, T. C., eds., *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*, 315–333. Springer.