`eclingo`: *A solver for Epistemic Logic Programs*∗

Pedro Cabalar[1], Jorge Fandinno[2], Javier Garea[1], Javier Romero[2] and Torsten Schaub[2]
[1] *University of Corunna, Spain.*
(*e-mail:* {cabalar,javier.garea}@udc.es)

[2] *University of Potsdam, Germany*
(*e-mail:* {fandinno,javier,torsten}@uni-potsdam.de)

submitted ; revised ; accepted

Abstract

We describe `eclingo`, a solver for epistemic logic programs under Gelfond 1991 semantics built upon the Answer Set Programming system `clingo`. The input language of `eclingo` uses the syntax extension capabilities of `clingo` to define subjective literals that, as usual in epistemic logic programs, allow for checking the truth of a regular literal in all or in some of the answer sets of a program. The `eclingo` solving process follows a guess and check strategy. It first generates potential truth values for subjective literals and, in a second step, it checks the obtained result with respect to the cautious and brave consequences of the program. This process is implemented using the multi-shot functionalities of `clingo`. We have also implemented some optimisations, aiming at reducing the search space and, therefore, increasing `eclingo`'s efficiency in some scenarios. Finally, we compare the efficiency of `eclingo` with two state-of-the-art solvers for epistemic logic programs on a pair of benchmark scenarios and show that `eclingo` generally outperforms their obtained results.

KEYWORDS: Answer Set Programming, Epistemic Logic Programs, Non-Monotonic Reasoning, Conformant Planning.

1 Introduction

The language of *epistemic specifications* (or epistemic logic programs), developed by Gelfond in three consecutive papers (Gelfond 1991; Gelfond and Przymusinska 1993; Gelfond 1994), is an extension of disjunctive logic programming that introduces modal constructs to quantify over the set of stable models (Gelfond and Lifschitz 1988) of a program. These new constructs, called *subjective literals*, have the form $\mathbf{K}\,l$ and $\mathbf{M}\,l$ where l is an *objective literal* l, that is, any atom p, its explicit negation -p, or any of these preceded by default negation. Intuitively, $\mathbf{K}\,l$ and $\mathbf{M}\,l$ respectively mean that l is true in every stable model (cautious consequence) or in some stable model (brave consequence) of the program. In many cases, these subjective literals can be seen as simple queries, but what makes them really interesting is their use in rule bodies, which may obviously affect the set of stable

∗ Partially supported by MINECO, Spain, grant TIC2017-84453-P. The second author is funded by the Alexander von Humboldt Foundation.

models they are quantifying. This feature makes them suitable for modelling introspection but, at the same time, may involve cyclic programs whose intuitive behaviour is not always easy to define. In general, the semantics of an epistemic logic program may yield alternative sets of stable models, each set being called a *world view*. For instance, the epistemic program

$$p \leftarrow not\ \mathbf{K}\, q \qquad\qquad q \leftarrow not\ \mathbf{K}\, p \tag{1}$$

yields two world views $\{\{p\}\}$ and $\{\{q\}\}$, each one with a single stable model. Deciding the intuitive world views of a cyclic program has motivated a wide debate in the literature. This was mostly due to the fact that Gelfond's original semantics (G91) manifests a kind of self-supportedness or *unfoundedness* typically illustrated by the epistemic program

$$p \leftarrow \mathbf{K}\, p \tag{2}$$

whose G91 world views are $\{\emptyset\}$ (as expected) but also $\{\{p\}\}$, which seems counterintuitive. Other semantics (Kahl et al. 2015; Fariñas del Cerro et al. 2015; Shen and Eiter 2017) managed to deal with this and other examples but fail to satisfy the elementary splitting property presented in (Cabalar et al. 2019b), something that was preserved by the original G91. Moreover, a first formalisation of foundedness was provided in (Cabalar et al. 2019a) and all the previously existing semantics violated that condition, except the new approach presented in that paper, FAEEL, which corresponds to a strengthening of G91 plus an extra foundedness check. Thus, to the best of our knowledge, FAEEL is the only semantics satisfying both splitting and foundedness up to date.

There exist several implemented solvers for epistemic logic programs – see (Leclerc and Kahl 2018) for a recent survey. Although there is no solver for FAEEL yet, the closest existing tools are those based on G91, since both semantics coincide in all epistemic logic programs whose subjective literals do not form positive cycles (Fandinno 2019). This suggests that a solver for FAEEL can be constructed by applying an extra foundedness check on top of a G91 solver. In fact, so far, all practical scenarios existing in the literature that involve epistemic problems can be represented without positive subjective cycles, so their computation in terms of G91 is sound with respect to FAEEL too.

In this paper, we present the `eclingo` system[1], a solver for epistemic logic programs under the G91 semantics. The tool is built on top of the ASP solver `clingo` (Gebser et al. 2019) making use of its features for syntactic extensions and multi-shot solving. The basic strategy applied by `eclingo` is a guess-and-check method where the truth value of subjective literals is first guessed with choice rules for auxiliary atoms and, in a second step, the obtained values for those atoms are checked using the sets of cautious and brave consequences of the program. This basic strategy has been improved with several optimisations. We have made experiments on several scenarios for a couple of benchmark domains and compared `eclingo` to `Wviews` (Kelly 2007) (another solver for G91) and `EP-ASP` (Son et al. 2017), which computes a close semantics (Kahl et al. 2015) also accepted by `eclingo`, and show that `eclingo` outperforms these tools in most cases.

The rest of the paper is organised as follows. In the next section, we recall the basic definition of G91 semantics for epistemic logic programs. In Section 3, we explain the input language of `eclingo` and illustrate its usage with an example. The next two sections

[1] `https://github.com/potassco/eclingo`

respectively explain the basic process and some implemented optimisations. Section 6 contains a comparison to solvers `Wviews` and `EP-ASP` on a pair of benchmark domains and, finally, Section 7 concludes the paper.

2 Background

We assume some familiarity with the answer set semantics (Gelfond and Lifschitz 1991) for logic programs. Given a set of atoms At, an *objective literal* is either an atom, a truth constant[2], that is, $a \in At \cup \{\top, \bot\}$, or an atom preceded by one or two occurrences of default negation, *not*. We assume that, for each atom $a \in At$, we have another atom '-a' in At that stands for the explicit negation of a. As usual, the answer sets of a standard program Π, denoted as AS[Π], are those stable models of Π that do not contain both a and -a. The syntax of epistemic logic programs is an extension of ASP. An expression of the form $\mathbf{K}\, l$, $\mathbf{M}\, l$ with l being an objective literal, is called *subjective atom*. A *subjective literal* can be a subjective atom A or its default negation $not\, A$. A *rule* is an expression of the form:

$$a_1 \vee \cdots \vee a_n \;\leftarrow\; L_1, \ldots, L_m \tag{3}$$

with $n \geq 0$ and $m \geq 0$, where each $a_i \in At$ is an atom and each L_j a literal, either objective or subjective. As usual, the left and right hand sides of (3) are respectively called the *head* and *body* of the rule. An *epistemic program* (or epistemic specification) is a set of rules. Given an epistemic program Π, we define $At(\Pi)$ as the set of all atoms that occur in program Π. Similarly, by $Heads(\Pi)$, we the denote the set of all atoms that occur in the head of any rule in Π and, by $Facts(\Pi)$, we denote the set of atoms that occur as facts in Π. Note that $Facts(\Pi) \subseteq Head(\Pi) \subseteq At(\Pi)$. Let $\mathbb{W}$ be a set of interpretations. We write $\mathbb{W} \models \mathbf{K}\, \ell$ if objective literal ℓ holds (under the usual meaning) in all interpretations of $\mathbb{W}$ and $\mathbb{W} \models not \mathbf{K}\, \ell$ if ℓ does not hold in some interpretation of $\mathbb{W}$.

Definition 1 (Subjective reduct)**.** *The* subjective reduct *of an epistemic program Π with respect to a set of propositional interpretations $\mathbb{W}$, written $\Pi^{\mathbb{W}}$, is obtained by replacing each subjective literal L by $\top$ if $\mathbb{W} \models L$ and by $\bot$ otherwise.* □

Note that the subjective reduct of an epistemic program does not contain subjective literals, and so, it is a standard logic program. Therefore, we can collect its answer sets AS[$\Pi^{\mathbb{W}}$]. We say that a set of propositional interpretations $\mathbb{W}$ is a *world view* of an epistemic program Π if $\mathbb{W} = \text{AS}[\Pi^{\mathbb{W}}]$. Early works on epistemic specifications allowed for empty world views $W = \emptyset$ when the program has no answer sets, rather than leaving the program without world views. Since this feature is not really essential, we exclusively refer to non-empty world views in this paper. The complexity for deciding whether an epistemic program has a world view is Σ_3^P (Truszczynski 2011), that is, one level higher in the polynomial hierarchy than the complexity of (disjunctive) ASP, which is Σ_2^P.

We conclude this section by introducing a well-known example from (Gelfond 1991).

[2] For a simpler description of program transformations, we allow truth constants where $\top$ denotes true and $\bot$ denotes false. These constants can be easily removed.

Example 1. *A given college uses the following set of rules to decide whether a student X is eligible for a scholarship:*

$$\mathit{eligible}(X) \leftarrow \mathit{high}(X) \tag{4}$$
$$\mathit{eligible}(X) \leftarrow \mathit{minority}(X), \mathit{fair}(X) \tag{5}$$
$$\text{-}\mathit{eligible}(X) \leftarrow \text{-}\mathit{fair}(X), \text{-}\mathit{high}(X) \tag{6}$$
$$\mathit{interview}(X) \leftarrow \mathrm{not}\,\mathbf{K}\,\mathit{eligible}(X), \mathrm{not}\,\mathbf{K}\,\text{-}\mathit{eligible}(X) \tag{7}$$

Here, $\mathit{high}(X)$ and $\mathit{fair}(X)$ refer to the grades of student X. The epistemic rule (7) *is encoding the college criterion "*The students whose eligibility is not determined by the college rules should be interviewed by the scholarship committee.*"* □

For instance, if the only available information for some student *mike* is the disjunction

$$\mathit{fair}(\mathit{mike}) \vee \mathit{high}(\mathit{mike}) \tag{8}$$

then the epistemic program (4)-(8) has a unique world view whose answer sets are:

$$\{\mathit{fair}(\mathit{mike}),\ \mathit{interview}(\mathit{mike})\} \tag{9}$$
$$\{\mathit{high}(\mathit{mike}),\ \mathit{eligible}(\mathit{mike}),\ \mathit{interview}(\mathit{mike})\} \tag{10}$$

3 Using `eclingo`

As said before, `eclingo` is based on `clingo`'s facilities (through its Python API) for syntax extension and multi-shot solving. As a result, `eclingo`'s input language is just a minor extension of the input language accepted by `gringo` (Gebser et al. 2009), the grounder used by `clingo`. In this way, `eclingo` programs can be constructed with three different types of statements: *rules*, *show* statements and *constant* definitions.

The structure of a `clingo` (or `eclingo`) rule is as follows:

$$H_1, \ldots, H_m \texttt{ :- } B_1, \ldots, B_n.$$

where the head is formed by `clingo` literals H_i and the body consists of elements B_i that can be `clingo` literals or subjective literals. As happens in sequent calculus, commas in the antecedent (the body) represent a conjunction whereas commas in the consequent (the head) represent a disjunction. The notation for subjective literals in `eclingo` is as follows. An expression $\mathbf{K}\,l$ is represented as a `clingo` theory atom `&k{`l`}` where l is a regular, objective literal, that is, it may combine an atom with default or explicit negation. The only particularity is that default negation *not* inside a `&k` operator must be replaced by the symbol $\sim$ (this limitation will be changed in the future). For instance, the subjective literal $\mathbf{K}$ *not -p* is currently represented in `eclingo` as `&k{~-p}`. Operator $\mathbf{M}$ is not directly supported but any literal $\mathbf{M}\,l$ can be represented as `not &k{~`l`}`.

For instance, program (1) is represented as the `eclingo` file `program1.lp`:

```
p :- not &k{q}.
q :- not &k{p}.
```

To obtain all world views of the program we just make the call

```
eclingo -n 0 program1.lp
```

getting the result:

```
eclingo version 0.2.0
Solving...
Answer: 1
&k{ p }
Answer: 2
&k{ q }
SATISFIABLE
```

Each answer provided by `eclingo` corresponds to some world view of the epistemic program Π, but expressed as the set X of those subjective literals that hold in the world view. The set X is enough to determine the answer sets in the world view. To see why, we can define the (non-epistemic) program Π_X where all subjective literals are replaced by their truth value $\top$ or $\bot$ with respect to X. The world view then just consists of the answer sets of Π_X. For instance, answer 1, makes `&k{p}` true and `&k{q}` false and, under that assumption, `program1.lp` produces the unique answer set `{p}`. We plan to include a future option to expand one or all world views into their sets of answer sets.

As with regular atoms in `clingo`, the input language of `eclingo` provides a `#show p/n` directive to select those subjective atoms that we want to be displayed in each world view. The syntax for this directive is the same as in `clingo`, where `p` is the name of some predicate (or its explicit negation) and `n` its arity, that is, the number of arguments. The difference in `eclingo` is that this directive refers to the predicates used in subjective atoms to be displayed in each world view. For instance, if we add the line

```
#show p/0.
```

to our previous example, the information for the second world view would just be empty, since we assume we only want to display subjective literals of the form `&k{p}`.

As a second example, the program consisting of (4)-(8) from Example 1 is represented in `eclingo` as:

```
eligible(X) :- high(X).
eligible(X) :- minority(X), fair(X).
-eligible(X) :- -fair(X), -high(X).
interview(X) :- not &k{ eligible(X)}, not &k{ -eligible(X)}, student(X).
student(mike).
fair(mike),high(mike).
```

where we have just introduced predicate `student` to make variable `X` safe in the epistemic rule for `interview`. The unique world view obtained by `eclingo` in this example shows an empty list of subjective atoms meaning that both `&k{eligible(mike)}` and `&k{-eligible(mike)}` are false. If we want to know what happens to predicate `interview` we can add the line:

```
#show interview/1.
```

to obtain now the output

```
Solving...
Answer: 1
&k{ interview(mike) }
SATISFIABLE
```

4 Basic solving process

As we have seen, `eclingo`'s input language is a minor modification of the one for `clingo`. This is possible thanks to the parsing methods of `clingo`'s API for obtaining the representation of an epistemic program as an abstract syntax tree (AST) in the form of a Python object. In that way, program transformations can be easily combined with the usual `clingo` functionality. The `eclingo` main algorithm solves epistemic logic programs following a guess and check strategy. In the guessing phase, subjective literals are replaced by auxiliary atoms and a regular logic program is generated. In the case of G91 semantics, this replacement of subjective literals is as follows. For each objective literal ℓ, we define its corresponding auxiliary atom aux_ℓ as:

$$aux_\ell \stackrel{\text{def}}{=} \begin{cases} aux\_p & \text{if } \ell = (p) \\ aux\_not\_p & \text{if } \ell = (\sim p) \\ aux\_sn\_p & \text{if } \ell = (-p) \\ aux\_not\_sn\_p & \text{if } \ell = (\sim -p) \end{cases}$$

for any atom p. Now, each positive subjective literal $\&\mathtt{k}\{\ell\}$ in the epistemic program is replaced by (`not not` aux_ℓ) whereas each negated subjective literal `not` $\&\mathtt{k}\{\ell\}$ is just replaced by (`not` aux_ℓ). Additionally, for each auxiliary atom aux_ℓ, we add the choice rule:

$$\{\ aux_\ell\ \}.$$

The result of this translation is a regular logic program that can now be used for guessing the truth values of subjective literals, represented as auxiliary atoms. Thus, when asking `clingo` to solve this program using its multi-shot feature, it will go returning answer sets that constitute potential *candidates* for world views. Since we only retrieve the auxiliary atoms aux_x from each answer set, we may have repeated answers (due to differences in the rest of atoms that are hidden). For this reason, we use the `clingo` "projection" option to rule out these duplicates.

In the checking phase, `eclingo` verifies that each candidate actually constitutes a valid world view. To this aim, we need to check several conditions on the subjective literals with respect to the answer sets of the candidate world view:

1. For each subjective literal $\&\mathtt{k}\{\ell\}$, literal ℓ must be in every answer set.
2. For each subjective literal `not` $\&\mathtt{k}\{\ell\}$, literal ℓ cannot be in every answer set.
3. For each subjective literal $\&\mathtt{k}\{\sim \ell\}$, literal ℓ cannot be in any answer set.
4. For each subjective literal `not` $\&\mathtt{k}\{\sim \ell\}$, literal ℓ must be in some answer set.

To obtain the answer sets of a candidate world view X, we would have to expand all the answers for Π_X provided by `clingo`. Fortunately, this expansion can be avoided since the four conditions above can be checked using `clingo` modes for *cautious* and *brave*

reasoning, that are computed by iterated intersection and union operations, respectively. Let $\mathit{cautious}(\Pi_X)$ and $\mathit{brave}(\Pi_X)$ denote the set of atoms in the cautious and brave consequences of Π_X, respectively. In particular, we can reduce those four conditions to:

1. For each subjective literal `&k{`ℓ`}`, check $\ell \in \mathit{cautious}(\Pi_X)$.
2. For each subjective literal `not &k{`ℓ`}`, check $\ell \notin \mathit{cautious}(\Pi_X)$.
3. For each subjective literal `&k{`$\sim \ell$`}`, check $\ell \notin \mathit{brave}(\Pi_X)$.
4. For each subjective literal `not &k{`$\sim \ell$`}`, check $\ell \in \mathit{brave}(\Pi_X)$.

Although `eclingo` was proposed as a solver for epistemic specifications under G91 semantics, it also supports the semantics proposed in (Kahl et al. 2015) (K15), which can be obtained by a simple transformation. In particular, K15 can be obtained from G91 by replacing each expression $\mathbf{K}\,\ell$ in the original epistemic program by the conjunction $\mathbf{K}\,\ell \wedge \ell$. When $\mathbf{K}\,\ell$ is not preceded by *not*, this simply generates an additional objective literal ℓ in the rule body. However, when we have to replace $\mathit{not}\,\mathbf{K}\,\ell$ by $\mathit{not}(\mathbf{K}\,\ell \wedge \ell)$, we obtain the disjunction $\mathit{not}\,\mathbf{K}\,\ell \vee \mathit{not}\,\ell$ that is replaced by a new auxiliary atom *aux* that is, then, defined by the pair of rules:

$$\begin{aligned} \mathit{aux} &\leftarrow \mathit{not}\,\mathbf{K}\,\ell \\ \mathit{aux} &\leftarrow \mathit{not}\,\ell \end{aligned}$$

Then, the rest of the translation proceeds as with G91.

5 Optimising the solving process

Several optimisations have been implemented on top of the basic solving process presented above. A first optimisation is the addition of *consistency constraints*. Notice that, once a subjective atom `&k{`ℓ`}` is replaced by a standard auxiliary atom *aux*, the relation to the original literal is lost. Thus, the guess may produce epistemically inconsistent combinations like, for instance, an answer set containing:

$$\{\texttt{\&k\{}\sim\texttt{p\}, p}\}$$

This problem can be avoided by adding the rule:

$$\texttt{:-}\ \mathit{aux}_\ell,\ \overline{\ell}.$$

for each subjective literal of the form `&k{`ℓ`}`, where $\overline{\ell} \stackrel{\text{def}}{=} \mathit{not}\,\ell$ if ℓ does not contain $\sim$ and $\overline{\sim\ \alpha} = \alpha$ if $\ell = (\sim \alpha)$. These constraints improve the efficiency of `eclingo` by ruling out epistemically inconsistent world views during the guessing phase, without requiring their subsequent check.

Another optimisation implemented in `eclingo` consists in using the grounder `gringo` to approximate the well-founded model (WFM) of the auxiliary guess program Π. Computing the WFM of a logic program takes a polynomial complexity in the general case, while computing the stable models of Π is Σ_2^P. This difference makes this heuristic a worthwhile strategy. The WFM of a program Π is a three-valued interpretation we can describe as a pair of disjoint sets of atoms $\langle I^+, I^-\rangle$ respectively collecting the true and false atoms in the model, being all the rest undefined. It is well-known that $I^+ \subseteq \mathit{cautious}(\Pi)$ and $I^- \subseteq \mathit{At} \setminus \mathit{brave}(\Pi)$. When `gringo` processes any program Π (even if it is originally ground) it performs some simplifications that allow retrieving

Algorithm 1: Extending an epistemic logic program using `gringo` grounder.

$F := \emptyset$; $H := At(\Pi)$;
$\Pi :=$ ground(Π);
while $(\mathit{Facts}(\Pi) \setminus F) \cap K^+ \neq \emptyset$ **or** $(H \setminus \mathit{Heads}(\Pi)) \cap K^- \neq \emptyset$ **do**
  foreach $p \in (\mathit{Facts}(\Pi) \setminus F) \cap K^+$ **do**
    $\Pi := \Pi \cup \{\texttt{aux\_p}\}$;
  end
  foreach $p \in (H \setminus \mathit{Heads}(\Pi)) \cap K^-$ **do**
    $\Pi := \Pi \cup \{\texttt{aux\_not\_p}\}$;
  end
  $F := \mathit{Facts}(\Pi)$; $H := \mathit{Heads}(\Pi)$;
  $\Pi :=$ ground(Π);
end

an approximation of the WFM $\langle I^+, I^-\rangle$ of Π. In particular, if Π' is the result provided by `gringo`, then $\mathit{Facts}(\Pi') \subseteq I^+$ and $At \setminus \mathit{Heads}(\Pi') \subseteq I^-$ which, in their turn, imply $\mathit{Facts}(\Pi') \subseteq \mathit{cautious}(\Pi)$ and $\mathit{Heads}(\Pi') \subseteq \mathit{brave}(\Pi)$. As a result, if we only use the grounder `gringo` on the guess program Π to obtain Π' we get a good estimate of regular atoms that are always true (resp. always false) in all the answer sets of the program. In particular, if we get $p \in \mathit{Facts}(\Pi')$ is true, then $\mathbf{K}\,p$ holds and we can safely add the corresponding auxiliary atom `aux_p`. The same happens if atom $p \notin \mathit{Heads}(\Pi')$ we can conclude $\mathbf{K}$ *not* p and add the auxiliary atom `aux_not_p`. These true subjective literals are added to the guess program Π to reduce the search space before computing the answer sets. Of course, this behaviour is implemented in an iterative way until no new addition is made, as described in Algorithm 1. Here, we use the set K^+ (resp. K^-) to collect every atom `p` occurring in some expression `&k{p}` (resp. `&k{~ p}`) in the original epistemic program. The algorithm uses two set variables, one for facts F and one for heads H, that are initially set to $\emptyset$ and $At(\Pi)$, respectively. Then, we repeat calls to `gringo`'s function ground(Π) while we obtain some new fact `p` originally used in a subjective literal `&k{p}` or we lose some head atom `p` originally used in a subjective literal `&k{~ p}`. When this happens, we include the corresponding auxiliary atoms in the program. To illustrate the algorithm, take Fig. 1 showing an `eclingo` input program on the left and its corresponding guess program Π_1 on the right. The latter generates 8

```
a :- not b.
c :- &k{~a}.
d :- not &k{~e}.
p :- &k{~d}.
```

```
a :- not b.
c :- aux_a.
d :- not aux_not_e.
p :- aux_not_d.
{aux_a}.
{aux_not_e}.
{aux_not_d}.
```

Fig. 1: An `eclingo` program (left) and its corresponding guess program Π_1 (right).

possible candidate world views corresponding to all the free combinations of truth values

for the auxiliary atoms. However, if we run `gringo` on Π_1 we obtain the new ground program Π_2:

```
a.
c :- aux_a.
d :- not aux_not_e.
p :- aux_not_d.
{aux_a}.
{aux_not_e}.
{aux_not_d}.
```

where $Facts(\Pi_2) = \{a\}$ and $Heads(\Pi_2)$ consists of $\{a, c, d, p\}$ and the auxiliary atoms. As a result, we know that $\mathbf{K}\, a$ and $\mathbf{K}$ *not* e must hold, and so, atoms `aux_a` and `aux_not_e` can be added to Π_2 or just replaced by $\top$ (`&true` in `gringo` notation). After doing that replacement on Π_2 if we run `gringo` again we obtain Π_3:

```
a.
c.
p :- aux_not_d.
{aux_not_d}.
```

but, as we can see, $d \notin Heads(\Pi_3)$ and we conclude $\mathbf{K}$ *not* d so atom `aux_not_d` can be replaced by $\top$. The resulting program has no auxiliary predicates and no new changes will occur after grounding, so the algorithm stops. In this example, the optimisation has solved the problem even before the guessing phase. This is because subjective literals where stratified in the original program. In the general case, however, the `gringo`-based optimisation is not so efficient if we have cyclic dependencies among the subjective literals.

6 Evaluation and comparison to other solvers

In this section, we compare `eclingo` with other two epistemic solvers, `Wviews` (Kelly 2007) and `EP-ASP` (Son et al. 2017), both with respect to usability and efficiency. The tool `Wviews`[3] is a solver for epistemic specifications under G91 semantics developed by Michael Kelly for his Honour's Thesis. It is built upon the ASP system `DLV` and allows the inclusion of subjective literals under a simple notation. However, this simplicity is eclipsed by the limitations in the rest of its grammar. `Wviews` parser is very sensitive to minor changes in the problem representation. In fact, we experienced problems to execute `Wviews` on programs with predicates with more than one argument, something that forced us to test the benchmarks for planning on ground programs. A peculiarity of `Wviews` is that it computes *all* the world views of an epistemic program.

`EP-ASP`[4] (Son et al. 2017) is another solver for epistemic logic programs that can compute world views under two semantics: (Kahl et al. 2015) (K15) (also computed by `eclingo`) and (Shen and Eiter 2017). Just like `eclingo`, it is built upon the ASP system

[3] `https://github.com/galactose/wviews`
[4] `https://github.com/tiep/EP-ASP`

clingo, but using version 4.5.3. EP-ASP input programs are generated using another independent and non-integrated tool, ELPS (Balai and Kahl 2014), that provides a method to translate an epistemic logic program with sort definitions into a standard ASP program. The grammar that defines a correct input program for ELPS is, therefore, substantially different from the one used by clingo. It considers four types of statements: directives, sort definitions, predicate declarations and rules. Directives can be either a constant definition or a maxint declaration, so the range for numerical expressions is predefined, unlike in clingo.

Regarding the efficiency comparison, we have executed the three tools (i.e., Wviews, EP-ASP and eclingo) on scenarios for two well-known problems in the literature: the Eligibility problem (Example 1) and a variant of the Yale Shooting problem with incomplete knowledge about the initial state, looking for a conformant plan (that is, a plan that always succeeds, regardless of the initial state). Experiments were performed on a machine equipped with an Intel i7-8550U (up to 4.0GHz) and 8GB memory running Ubuntu 18.04.4 LTS. The times were measured using a Python wrapper and taking the average of 10 different executions for each problem instance. Encodings for eclingo scenarios can be downloaded from the Git repository. It is important to note that EP-ASP encodings are already preprocessed by ELPS, so our execution times do not consider this translation step.

Table 1 shows the average times for the Scholarship Eligibility Problem obtained by Wviews under G91 semantics, EP-ASP under K15 semantics and eclingo under both G91 and K15 semantics, to make a fair comparison. In this problem, the tools were fed with 25 scenarios denoted as eligibleXX were XX is the number of the instance and, at the same time, the number of students for the problem. As can be seen, Wviews can only solve the first 8 scenarios (with a timeout of 2 minutes) and only performs better than eclingo in the first one. The performance of eclingo is more robust, solving 21 instances clearly below 1 second, and showing a slight grow (up to 3.35s) for the 4 larger instances. Note that eclingo is computing all the world views of the epistemic program, since this is default mode for Wviews. However, in the comparison with EP-ASP (the last three columns) we just look for one world view. The options we used for that solver were pre=1, max=0 (brave/cautious preprocessing, and K15 semantics, respectively). EP-ASP solves 16 scenarios and reaches the timeout of 120s for the 9 remaining ones. For the first 9 scenarios (with the exception of eligible06), EP-ASP execution times are better but very close to eclingo ones. However, in the examples from 10 to 16 the performance of EP-ASP is clearly worse and, moreover, shows an unpredictable variability among the solved cases, from 0.063s in eligible15 and eligible16 to 44.96s for eligible12. eclingo under K15 solves the 25 scenarios in a range of times from 0.03s to 0.05s, except eligible25 that just takes 0.54s. When using G91 semantics, we get the same the world views (for this domain) but the eclingo times are slightly better, since K15 is computed as a translation to G91.

For the Yale shooting benchmark we actually extended the benchmarks from the EP-ASP repository with the instances yale09 to yale13, all for path length 10 and the last one without any world view, to try an unsatisfiable problem. Table 2 shows average execution times obtained by EP-ASP under K15 semantics and eclingo under G91 semantics. In this case, the comparison is less accurate than the eligibility example for several reasons. First, we have used two slightly different problem encodings. For EP-ASP,

	Computing all world views		Computing one world view		
	`Wviews G91`	`eclingo G91`	`EP-ASP K15`	`eclingo K15`	`eclingo G91`
`eligible01`	**0.025**	0.035	**0.024**	0.033	0.034
`eligible02`	0.042	**0.036**	**0.021**	0.034	0.035
`eligible03`	0.103	**0.035**	**0.022**	0.035	0.033
`eligible04`	0.347	**0.036**	**0.023**	0.036	0.034
`eligible05`	1.397	**0.036**	**0.025**	0.035	0.034
`eligible06`	5.728	**0.037**	0.138	0.037	**0.035**
`eligible07`	27.271	**0.037**	**0.031**	0.037	0.036
`eligible08`	113.188	**0.037**	**0.032**	0.037	0.036
`eligible09`	-	**0.039**	**0.035**	0.039	0.037
`eligible10`	-	**0.041**	1.795	0.039	**0.037**
`eligible11`	-	**0.048**	12.302	0.041	**0.040**
`eligible12`	-	**0.049**	44.959	0.043	**0.040**
`eligible13`	-	**0.049**	0.934	0.044	**0.041**
`eligible14`	-	**0.050**	13.574	0.045	**0.041**
`eligible15`	-	**0.048**	0.063	0.044	**0.042**
`eligible16`	-	**0.085**	0.063	0.054	**0.040**
`eligible17`	-	**0.150**	-	0.041	**0.040**
`eligible18`	-	**0.142**	-	0.043	**0.041**
`eligible19`	-	**0.392**	-	0.050	**0.042**
`eligible20`	-	**0.414**	-	0.049	**0.041**
`eligible21`	-	**0.567**	-	0.049	**0.042**
`eligible22`	-	**1.516**	-	0.049	**0.043**
`eligible23`	-	**1.015**	-	0.051	**0.044**
`eligible24`	-	**0.937**	-	0.050	**0.044**
`eligible25`	-	**3.347**	-	0.544	**0.048**

Table 1: Eligibility Problem. Time in seconds: timeout fixed in 120s.

we used K15 semantics on the benchmarks provided with the tool, already translated from `ELPS`. Moreover, we used the recommended `EP-ASP` configuration for planning and heuristics, passing the options `pre=1, max=0, planning=1, heuristic=1`. In this special configuration, `EP-ASP` recognizes the action theory representation (fluents, actions, goals, etc) and is capable of applying planning based heuristics. In particular, `EP-ASP` solves the problem first as a regular planning domain and then uses this result to prune the search space for the conformant planning problem. For `eclingo`, we redesigned the epistemic rules in the scenario to use G91 instead, something that, under our understanding, provides a more natural use of the epistemic operators (see Cabalar et al. 2019b for a discussion). Besides, `eclingo` does not apply any planning-based specific heuristics or optimisation: it treats all the scenarios as regular epistemic specifications[5].

As we can see in Table 2, `EP-ASP` in planning mode performs slightly better than `eclingo` in all the cases solved by both tools. However, `EP-ASP` reaches the timeout in three scenarios for path length 10, while `eclingo` solves all of them.

Due to lack of encodings and the difficulty shown by `Wviews` grammar to represent the epistemic Yale Shooting Problem, we generated a ground version of the problem. Thus, both `Wviews` and `eclingo` are compared under this ground program. Table 3 shows the average times for 10 runs of 12 scenarios of the problem. As we can see, `Wviews` can only

[5] We also executed `EP-ASP` for these benchmarks without using the planning mode, but it produced apparently incoherent results, immediately printing an empty world view.

Computing one world view			
	`EP-ASP K15`, planning mode	`eclingo G91`	Path length
`yale01`	**0.025**	0.042	1
`yale02`	**0.024**	0.039	2
`yale03`	**0.027**	0.040	3
`yale04`	**0.027**	0.040	4
`yale05`	**0.033**	0.045	5
`yale07`	**0.042**	0.073	7
`yale08`	**0.036**	0.051	8
`yale09`	-	**0.314**	10
`yale10`	**0.068**	0.257	10
`yale11`	-	**0.106**	10
`yale12`	**0.119**	1.013	10
`yale13`*	-	**0.445**	10

Table 2: Yale Shooting Problem. Time in seconds: timeout fixed in 120s. Problem yale13 is unsatisfiable.

solve the first three scenarios while `eclingo` can still solve all of them, although with worse execution times than `eclingo` in the non-ground version (Table 2). This is because the independent grounding we used is apparently less efficient, generating more ground subjective literals and creating harder instances.

Computing all world views			
	`Wviews G91`	`eclingo G91`	Path length
`ground_yale01`	0.054	**0.038**	1
`ground_yale02`	0.590	**0.040**	2
`ground_yale03`	11.330	**0.042**	3
`ground_yale04`	-	**0.046**	4
`ground_yale05`	-	**0.063**	5
`ground_yale07`	-	**0.230**	7
`ground_yale08`	-	**0.108**	8
`ground_yale09`	-	**2.137**	10
`ground_yale10`	-	**25.521**	10
`ground_yale11`	-	**28.702**	10
`ground_yale12`	-	**59.013**	10
`ground_yale13`*	-	**2.278**	10

Table 3: Ground version of the Yale Shooting Problem. Time in seconds: timeout fixed in 120s. Problem ground_yale13 is unsatisfiable.

7 Conclusions and Related Work

We have presented `eclingo`, a solver for epistemic specifications under G91 semantics. The solver is programmed on top of `clingo`, using its syntactic extension and multi-shot solving features. We have tested the execution of `eclingo` and compared to other two epistemic solvers in a pair of domains from the literature. The results seem to point out that `eclingo` provides a better performance, especially in the number of solved scenarios.

Our future work includes the addition of other optimisation techniques and, more importantly, the implementation of an unfoundedness check to disregard self-supported

world views that are sometimes produced by G91 semantics (although only on positive cycles), computing in this way the stronger semantics provided in (Cabalar et al. 2019a). We also plan to extend the benchmarks with harder instances that can be parameterised and possibly include comparisons to solvers under other semantics, on scenarios where it can be guaranteed that the same solutions are obtained.

References

Balai, E. and Kahl, P. 2014. Epistemic logic programs with sorts. In *Proceedings of the Workshop on Answer Set Programming and Other Computing Paradigms, 2014*, D. Inclezan and M. Maratea, Eds.

Balduccini, M., Lierler, Y., and Woltran, S., Eds. 2019. *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19).* Lecture Notes in Artificial Intelligence, vol. 11481. Springer-Verlag.

Cabalar, P., Fandinno, J., and Fariñas del Cerro, L. 2019a. Founded world views with autoepistemic equilibrium logic. See Balduccini et al. (2019), 134–147.

Cabalar, P., Fandinno, J., and Fariñas del Cerro, L. 2019b. Splitting epistemic logic programs. See Balduccini et al. (2019), 120–133.

Fandinno, J. 2019. Founded (auto)epistemic equilibrium logic satisfies epistemic splitting. *Theory and Practice of Logic Programming 19,* 5-6, 671–687.

Fariñas del Cerro, L., Herzig, A., and Iraz Su, E. 2015. Epistemic equilibrium logic. In *Proceedings of the Twenty-fourth International Joint Conference on Artificial Intelligence (IJCAI'15)*, Q. Yang and M. Wooldridge, Eds. AAAI Press, 2964–2970.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming 19,* 1, 27–82.

Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., and Thiele, S. 2009. On the input language of ASP grounder gringo. In *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Artificial Intelligence, vol. 5753. Springer-Verlag, 502–508.

Gelfond, M. 1991. Strong introspection. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91)*, T. Dean and K. McKeown, Eds. AAAI Press / The MIT Press, 386–391.

Gelfond, M. 1994. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence 12,* 1-2, 89–116.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Gelfond, M. and Przymusinska, H. 1993. Reasoning on open domains. In *Logic Programming and Non-monotonic Reasoning, Proceedings of the Second International Workshop, Lisbon, Portugal, June 1993*, L. Moniz Pereira and A. Nerode, Eds. MIT Press, 397–413.

Kahl, P., Watson, R., Balai, E., Gelfond, M., and Zhang, Y. 2015. The language of epistemic specifications (refined) including a prototype solver. *Journal of Logic and Computation*.

Kelly, M. 2007. Wviews: A worldview solver for epistemic logic programs. Honour's thesis (supervised by Yan Zhang). University of Western Sydney.

Leclerc, A. and Kahl, P. 2018. A survey of advances in epistemic logic program solvers. In *Proceedings of the Eleventh International Workshop on Answer Set Programming and other Computer Paradigms (ASPOCP'18).*

Shen, Y. and Eiter, T. 2017. Evaluating epistemic negation in answer set programming (extended abstract). In *Proceedings of the Twenty-sixth International Joint Conference on Artificial Intelligence (IJCAI'17)*, C. Sierra, Ed. IJCAI/AAAI Press, 5060–5064.

Son, T. C., Le, T., Kahl, P. T., and Leclerc, A. P. 2017. On computing world views of epistemic logic programs. In *Proc. of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 1269–1275.

Truszczynski, M. 2011. Revisiting epistemic specifications. M. Balduccini and T. Son, Eds. Lecture Notes in Computer Science, vol. 6565. Springer, 315–333.