ECAI 2025
I. Lynce et al. (Eds.)

doi:10.3233/FAIA250970

A Prototyping Framework for Reduct-Based ELP Solvers: Methodology and Implementation

Stefania Costantini [a,*,1] **and Andrea Formisano**[b]

[a]DISIM - University of L'Aquila and GNCS - INdAM, Italy
[b]DMIF - University of Udine and GNCS - INdAM, Italy
ORCID (Stefania Costantini): https://orcid.org/0000-0002-5686-6124, ORCID (Andrea Formisano): https://orcid.org/0000-0002-6755-9314

Abstract. Epistemic Logic Programs (ELPs) extend Answer Set Programming (ASP) by incorporating epistemic operators, notably the knowledge operator **K**. The semantics of ELPs are defined through the concept of *world views*, which are sets that, in turn, comprise sets of atoms. Various semantic frameworks have been proposed, many of which, including influential early approaches, are based on *reduct-based* definitions. These approaches generalize the ASP methodology to ELPs by selecting a candidate world view, constructing the corresponding *reduct* of the program based on this candidate, computing the stable models of this reduct, and subsequently verifying if the initial candidate is indeed a world view. While specialized inference engines (ELP solvers) have been developed for certain semantic approaches, there remains no consensus regarding the "correct" semantics for ELPs, and new or variant semantics continue to emerge. In response to this evolving situation, this paper introduces a novel fast prototyping methodology that enables the implementation of solvers for any reduct-based semantics. The main advantage of this approach is the ability to rapidly experiment with new semantics on small- to medium-sized programs, as opposed to the very limited small-scale experimentation seen in the literature. This facilitates a thorough preliminary evaluation prior to committing to the more resource-intensive development of dedicated solvers. As a concrete demonstration, we apply our methodology to seminal semantic frameworks already established in the literature. Nevertheless, our approach is readily adaptable to accommodate other reduct-based semantics.

1 Introduction

Epistemic Logic Programs (ELPs), hereafter referred to simply as *programs* unless otherwise specified, were originally introduced by [17] and [14]. The primary goal was to straightforwardly extend Answer Set Programming (ASP), which is based on the Answer Set Semantics [16], with *epistemic operators*. These operators enable epistemic reasoning in a practical yet principled manner. Since their inception, there has been interest in enhancing ASP solvers—specialized inference engines for computing the answer sets of given ASP programs—to handle ELPs.

However, developing solvers capable of managing ELPs proved challenging, largely due to the inherent complexity of their semantics. The epistemic operators introduced in ELPs allow introspective reasoning about program's semantics itself, which are defined in terms of answer sets. Central to Epistemic Logic Programming is the epistemic operator **K**, symbolizing *knowledge*, where $\mathbf{K}A$ is considered true if the atom A is true in every answer set of the program Π containing $\mathbf{K}A$. This notion naturally extends to literals as well. The derived operator of *possibility*, denoted by $\mathbf{M}A$, indicates that the atom A is true in at least one of the answer sets of Π. Additionally, the derived *epistemic negation operator*, denoted by **not** A, expresses that A *is not provably true*, meaning that A is false in at least one answer set of Π. Formally, $\mathbf{M}A$ and **not** A correspond to the conditions $not\ \mathbf{K}\ not A$ and $not\ \mathbf{K}\ A$, resp., where not represents standard ASP default negation.

Semantics of ELPs is provided in terms of some mechanism to characterize *world views*, which are sets of answer sets (instead of a unique set of answer sets like in Answer Set Programming). Among the several semantic approaches that have been introduced for ELPs beyond the seminal ones, we mention those defined in [15, 26, 12, 24, 19, 25, 3].

Many of these approaches utilize a *reduct-based* methodology, extending the established practice from ASP to ELPs. Specifically, to determine the world views of a given program, they involve the following steps: (i) select an initial candidate world view; (ii) following a particular semantic definition, construct the *reduct* of the program based on the chosen candidate, which is a new ASP program; (iii) compute the stable models of this reduct; (iv) verify whether the initial candidate actually constitutes a valid world view.

Some of these reduction-based approaches are supported by solvers (see [21] for a survey), but these have usually been developed to handle a single semantics. However, the lack of consensus on the "correct" semantics for ELPs has led to the introduction of new semantic variants —and it is likely that more will continue to emerge in the future. To facilitate experimentation with novel semantics, we propose a fast-prototyping methodology for constructing solvers tailored to any reduct-based semantics. This approach enables researchers to test new ideas on small programs, overcoming the traditional limitation to only tiny examples. As such, it postpones the need for the costly development of optimized, dedicated solvers until there is sufficient justification.

To validate the practicality of our approach, we implemented it within the ASP Chef system [1] and conducted experiments on all examples available in the literature, along with additional test cases.

* Corresponding Author. Email: stefania.costantini@univaq.it
[1] Equal contribution.

The paper is structured as follows. Section 2 provides a brief overview of ASP. Section 3 offers a summary of the ELPs, highlighting their potential applications, anticipated semantic features, and the semantics introduced in [14]. Section 4 presents a concise description of the ASP Chef system. In Sections 5 and 6, we introduce our fast-prototyping approach for constructing a solver based on any reduct-based semantics, focusing on ease of development rather than efficiency. These sections detail both the underlying method and its realization within the ASP Chef system. Finally, Section 7 concludes the paper.

2 Answer Set Programming in a Nutshell

In ASP [2], which is grounded in the Answer Set Semantics [16], a program is viewed as a set of declarative statements that define a problem, and consists of a collection of *rules* of the form:

$$A_1 \vee \ldots \vee A_g \leftarrow L_1, \ldots, L_n \tag{1}$$

Here, each A_i $(0 \leq i \leq g)$ is an atom, $\vee$ denotes disjunction, and each L_i $(0 \leq i \leq n)$ is a literal. A literal is either an atom or a default-negated atom of the form $not\ A$, where not represents default negation. The left-hand side of the rule is referred to as the *head*, and the right-hand side as the *body*.

A rule with an empty body is known as a *fact*. Disjunction is permitted only in the heads of rules, including facts. A rule with an empty head, or equivalently a head represented by $\bot$, takes the form $\leftarrow L_1, \ldots, L_n$ or $\bot \leftarrow L_1, \ldots, L_n$ and is called a *constraint*. Such rules express that the conjunction $L_1, \ldots, L_n$ must not hold simultaneously in any answer set.

Each *answer set* corresponds to a solution that satisfies this specification. If a program has no answer sets, it is considered *inconsistent*; otherwise, it is *consistent*. The presence of constraints often plays a key role in rendering a program inconsistent.

Several extensions of ASP have been proposed (such as those involving aggregates, weak constraints, etc.). We refer the reader to [13, 10] and the references therein for a detailed description.

This paper implicitly assumes all atoms as propositional (namely, we consider *ground* programs only).

The answer set (or *stable model*) semantics can be defined through multiple equivalent formulations [22, 6]. Conceptually, the answer sets of a program Π, when they exist, are among the supported minimal classical models of Π, understood as a first-order theory in a natural way.

The original definition, introduced by [16] for programs with rule heads restricted to single atoms, is based on the *Gelfond–Lifschitz (GL) operator*. Given a program Π and a set of atoms I, the GL-operator constructs the *reduct* Π^I by removing all rules whose bodies contain a default-negated atom $not\ A$ such that $A \in I$, and then removing all default negations from the remaining rules. The result is a definite (i.e., negation-free) logic program. The operator then maps I to the least model of Π^I. A set of atoms I is an *answer set* of Π if and only if it is a fixed point of this operator, i.e., I is the least model of its own reduct Π^I.

The different answer sets of Π represent the different possible alternative state of affairs (in terms of what is true and what is not) which are compatible with the problem specification expressed in Π, and from which it is possible to extract solutions to the problem.

Well-developed freely available *answer set solvers* exist that compute the answer sets of a given program [20, 27]. Solvers are normally provided within engines that feature auxiliary functionalities to aid programmers.

3 Epistemic Logic Programs and their Semantics

3.1 Introduction and Intuition

In Epistemic Logic Programming (ELP), one can postulate about what is known, meaning true in every answer set of a program, in the program itself. This is done through literals of the form $\mathbf{K}L$, $\mathbf{K}$ intuitively meaning "knowledge", where $\mathbf{K}L$ is called a *subjective literal* in contrast to usual *objective literals*.

Note that mentioning L in some rules of a given program Π means that L is intended to be true in some of the answer sets of Π. In contrast, $\mathbf{K}L$ has a much stronger connotation, meaning that L is unequivocally true in any situation, i.e., true in all the answer sets. This makes Epistemic Logic Programming suitable for all those critical applications in fields such as law, cybersecurity, distributed computing, recommender systems, etc., where one must be able to refer to, so to say, *reliable truth*.

Still, as in ASP, uncertainty or conflict about the truth of some atom gives rise to several answer sets; in addition, in ELP, uncertainty or conflict about knowledge gives rise to radically alternative scenarios called *world views*, that are sets of answer sets, each one stemming from the given program and the assumptions on knowledge expressed therein. For example, the program

$$\begin{array}{l} a \leftarrow not\ b \\ b \leftarrow not\ a \\ e \leftarrow not\ \mathbf{K}f \\ f \leftarrow not\ \mathbf{K}e \end{array} \tag{2}$$

has two world views, under every semantics: one is $[\{a, e\}, \{b, e\}]$, where $\mathbf{K}e$ is true (as e is, in fact, true in both answer sets) and $\mathbf{K}f$ is false, and the other one is $[\{a, f\}, \{b, f\}]$ where $\mathbf{K}f$ is true (as f is, in fact, true in both answer sets) and $\mathbf{K}e$ is false. Note that, according to a widely-used convention, each world view, which is a set of answer sets, is denoted by using brackets $[\,]$. The presence of two answer sets in each world view of the above program is due to the cycle on objective atoms a and b. In contrast, the presence of two world views is due to the cycle involving subjective literals (in general, the existence and number of world views are related to such kinds of cycles; see [5] for a detailed discussion).

As a concrete example, consider the following program, expressed in basic non-disjunctive ASP enriched via the $\mathbf{K}$ operator. The program states (in a hypersimplified way) under which conditions a patient should consult a doctor. In particular, in the following formulation, a patient will consult a specific doctor for some problem p on which the doctor is specialized only if the doctor is known to be reliable. Notice here the difference that this brings to the formulation: it is not sufficient that the doctor might be possibly reliable (in some answer set) but must be certainly reliable (in every answer set). Since the first rule must be grounded, in order to show concise world views we consider below only one doctor.

The initial version of the program can be the following:

$$\begin{array}{l} consult(patient, X, p) \leftarrow doctor(X), specialized(X, p), \\ \qquad\qquad\qquad\qquad\quad good\_reputation(X), \\ \qquad\qquad\qquad\qquad\quad \mathbf{K}\, reliable(X). \\ doctor(d1). \\ specialized(d1, p). \end{array}$$

This embryonic program has only one world view, shown below, with just one answer set inside (the world view is enclosed in square brackets and the answer set in braces):

$$[\{doctor(d1), specialized(d1, p)\}]$$

There is no indication in the program, and thus, from the world view, that the doctor is reliable and has a good reputation and therefore can be consulted. One can then extend the program to outline radical uncertainty by stating that a doctor may be known to be either reliable or unreliable. This is expressed by a negative cycle on *not* **K**, that generates two world views.

$$reliable(X) \leftarrow \ doctor(X), not\, \mathbf{K}\, unreliable(X).$$
$$unreliable(X) \leftarrow \ doctor(X), not\, \mathbf{K}\, reliable(X).$$

Assume that there is also uncertainty about the doctor's good reputation, expressed in non-disjunctive ASP as follows:

$$good\_reputation(d1) \leftarrow \ not\, nogood\_reputation(d1).$$
$$nogood\_reputation(d1) \leftarrow \ not\, good\_reputation(d1).$$

The program outlined so far has, in fact, two world views, each including two answer sets, reflecting the uncertainty about the doctor's good reputation. The doctor is concluded reliable in one world view, $W1$, and unreliable in the other one, $W2$. Still, even in the former world view it is not concluded that (s)he can be consulted because of the uncertainty about her reputation.

$$\begin{aligned} W1 = \ & [\{doctor(d1), specialized(d1,p), \\ & reliable(d1), good\_reputation(d1)\}, \\ & \{doctor(d1), specialized(d1,p), \\ & reliable(d1), nogood\_reputation(d1)\}], \end{aligned}$$

$$\begin{aligned} W2 = \ & [\{doctor(d1), specialized(d1,p), \\ & unreliable(d1), good\_reputation(d1)\}, \\ & \{doctor(d1), specialized(d1,p), \\ & unreliable(d1), nogood\_reputation(d1)\}]. \end{aligned}$$

Assume now that a new rule will be established stating that a doctor has a good reputation if (s)he is known to have issued brilliant diagnoses in the past and that evidence about this will also be acquired concerning doctor $d1$.

$$\begin{aligned} good\_reputation(X) \leftarrow \ & doctor(X), \\ & \mathbf{K}\, past\_brilliant\_diagnoses(X). \end{aligned}$$
$$past\_brilliant\_diagnoses(d1).$$

Note that $\mathbf{K}\, past\_brilliant\_diagnoses(d1)$ immediately follows as its argument is a fact, i.e., is unconditionally true. The conclusion $good\_reputation(X)$ that can be now drawn rules out the second answer set from both world views, which become:

$$\begin{aligned} W1' = \ & [\{doctor(d1), specialized(d1,p), \\ & reliable(d1), good\_reputation(d1), \\ & past\_brilliant\_diagnoses(d1), \\ & consult(patient, d1, p)\}], \end{aligned}$$

$$\begin{aligned} W2' = \ & [\{doctor(d1), specialized(d1,p), \\ & unreliable(d1), good\_reputation(d1), \\ & past\_brilliant\_diagnoses(d1)\}]. \end{aligned}$$

Therefore, according to $W1$, the patient can finally consult the doctor about her problem. In addition, since a brilliant doctor can hardly be considered unreliable, one might add the following constraint, stating, in fact, that it is impossible that a doctor issuing brilliant diagnoses in the past is known to be unreliable:

$$\begin{aligned} \leftarrow \ & doctor(X), \\ & \mathbf{K}\, past\_brilliant\_diagnoses(X), \mathbf{K}\, unreliable(X). \end{aligned}$$

This constraint rules out the second world view, so $W1'$ will become the unique world view of the program, and no uncertainty is left.

Notice that for this very simple program that we have incrementally constructed, we end up with a single world view, but this is not always the case. In general, there can be several world views, and so uncertainty exists about which scenario (outlined in a world view) is more reliable. Extensions to the basic ELP approach (that we do not consider here) provide epistemic operators ranging over the entire set of world views.

3.2 Syntax and Semantics

Epistemic Logic Programs allow one to express positive and negative *subjective literals* (in addition to *objective literals*, which are those that can occur in plain ASP programs). A positive subjective literal is of the form $\mathbf{K}L$, where **K** is the *epistemic operator* of knowledge and L is an objective literal (hence, epistemic operators cannot be nested), meaning that L is "known" as it is true in all answer sets of the given program Π (namely, L is a *cautious consequence* of Π).

The syntax of rules in an ELP program is analogous to ASP, save that literals in the body can now be either objective or subjective.

An ELP program is called *objective* if no subjective literals occur therein; that is, it is an ASP program. A constraint involving (also) subjective literals is called a *subjective constraint*, whereas one involving objective literals only is an *objective constraint*.

Let At be the set of atoms occurring (within either objective or subjective literals) in a given program Π, and $Atoms(r)$ be the set of atoms occurring in rule r. Let $Head(r)$ be the head of rule r and $Body_{obj}(r)$ (resp., $Body_{subj}(r)$) be the (possibly empty) set of objective (resp., subjective) literals occurring in the body of r. One often writes $Head(r)$ and $Body_{obj}(r)$ in place of $Atoms(Head(r))$ and $Atoms(Body_{obj}(r))$, respectively, when the intended meaning is clear from the context. *Subjective rules* are those whose body is only composed of subjective literals.

Let a semantics $\mathcal{S}$ be a function mapping each program into sets of belief views, that is sets of sets of objective literals, where $\mathcal{S}$ has the property that, if Π is an objective program, then the unique member of $\mathcal{S}(\Pi)$ is the set of stable models of Π. Given a program Π, each member of $\mathcal{S}(\Pi)$ is called an *$\mathcal{S}$-world view* of Π. One usually writes "world view" instead of "$\mathcal{S}$-world view" whenever mentioning the specific semantics is irrelevant.

As usual, for any world view W and any subjective literal $\mathbf{K}L$, we write $W \models \mathbf{K}L$ if and only if for all $I \in W$ the literal L is satisfied by I (i.e., if $L \in I$ for L atom, or $A \notin I$ if L is *not* A). W satisfies a rule r if each $I \in W$ satisfies r.

A noteworthy effort to identify desirable semantic properties for ELPs can be found in the work of Cabalar et al. [4, 3]. The authors suggest that several properties originally developed for ASP could also be beneficial when adapted to the context of ELPs.

One such key property is *foundedness*, which, intuitively, ensures that the conclusions drawn in a world view are actually supported by the program. Specifically, foundedness prevents atoms within a world view from being derived solely through cyclic positive dependencies. In this context, $\mathbf{K}A$ is treated as equivalent to A when identifying such cycles. This property strengthens the semantic robustness of a world view by disallowing self-supporting derivations. Another important notion is that of *epistemically stratified programs* [3, 7]. These programs, by virtue of a well-structured use of epistemic operators, admit a unique world view. Stratification ensures a clear hierarchy in epistemic reasoning, eliminating ambiguity in interpretation.

The first formal semantics for ELPs –here referred to as *G94*– was introduced in the seminal work of [14] and is reported below.

In the following, let Π denote an ELP program and r a rule in Π.

Definition 1 (G94-world views). *The G94-reduct of* Π *with respect to a non-empty set of interpretations* W *is obtained by:*

(i) replacing by $\top$ *every subjective literal* $L \in Body_{subj}(r)$ *such that* L *is of the form* $\mathbf{K}G$ *and* $W \models G$*, and*
(ii) replacing all other occurrences of subjective literals of the form $\mathbf{K}G$ *by* $\bot$*.*

A non-empty set of interpretations W *is a G94-world view of* Π *if and only if* W *coincides with the set of all stable models of the G94-reduct of* Π *with respect to* W*.*

This semantics is important because it is easy to compute and because it is related to FAAEL [3], a well-established recent semantic approach whose definition is not reduct-based. Rather, it is based on the modal logic KD45 where a belief view is transformed from a set of interpretations to a set of HT-interpretations, that is, interpretations in terms of the logic of Here-and-There (HT) [23]. It turns out, in fact, that FAAEL world views are exactly the founded G94 world views (as G94 does not enjoy foundedness). This provides a way to implement FAAEL, as a solver for G94 indeed exists.

The reader may refer to [21] for a list of existing solvers for ELPs, each one referring to a single semantic approach. The method that we propose below to implement a solver is instead semantic-agnostic and customizable to any reduct-based approach.

4 The ASP Chef system

In Section 5, we will describe an approach to quick prototyping of ELP solvers, and in Section 6, we will describe its implementation in the ASP Chef system [1]. In preparation for this, let us first provide some background on ASP Chef.

ASP, as observed in [1], has been widely applied to complex search and optimization combinatorial problems in real-world and industrial applications [8, 9, 11], features linguistic high-level constructs typical of logic-based programming languages but is not formulated nor intended as a comprehensive programming language. Consequently, ASP is best utilized to tackle particular tasks within a more extensive pipeline. This implies that tasks managed by ASP engines are anticipated to receive input from other modules within the pipeline, potentially implemented using diverse paradigms. Likewise, these tasks are also expected to generate output for consumption by subsequent modules in the pipeline, which may also employ varied paradigms.

The ASP Chef system that we will use in our implementation aims to provide a framework, in which users can employ ASP directly by specifying a set of logic rules but also indirectly via mapping data from one format to the format accepted by ASP engines and mapping the output produced by ASP engines to some other format suitable to be presented to the end-user or further processed in a pipeline. ASP Chef is thus designed to ease the creation of operations pipelines rather than being just an IDE (Integrated Development Environment) or an editor for ASP.

In the ASP Chef system, there is a notion of ASP *recipe* as a chain of *ingredients* that are the instantiation of different operations, where an operation can be one of the "traditional" ASP computational tasks, or some data manipulation procedure or data visualization procedure. To enable the composition of linear chains of ingredients, sequences of interpretations (i.e., sequences of sets of atoms) have been adopted as a uniform format for the input and the output of all operations that can have parameters to customize their behaviour and have side output to enable inspection and visualization of intermediate states of the evaluation of ASP recipes. The adopted uniform format allows several operations implemented in ASP Chef to be combined in any order and new operations to be easily accommodated.

A web app (https://asp-chef.alviano.net/) is available to implement even long pipelines involving ASP as a core engine to perform several computational tasks, putting into practice the notion of ASP recipe. Several operations are already available; the default ASP engine exploited in ASP Chef is `clingo-wasm`, a web-accessible version of `clingo` [27].

Operations that have already been implemented and are available to future developers include searching whether an atom is in an answer set and filtering optimal answer sets according to a given objective function, sorting atoms within each model according to their lexicographical ordering, merging and splitting interpretations.

5 Fast prototyping of a solver: the Method

Let us consider a given semantics $\mathcal{S}$ for ELP based on a notion $\mathcal{R}$ of reduct (e.g., the G94 semantics introduced by Def. 1). To check whether an epistemic interpretation $\mathcal{W}$ is an $\mathcal{S}$-world view for program Π, one has to apply $\mathcal{R}$ to Π to obtain the reduced program $\Pi_{\mathcal{R}}$. Next, one has to find the set $\mathcal{W}'$ of answer sets of $\Pi_{\mathcal{R}}$. At this point, depending on the specific $\mathcal{S}$, some form of post-processing $\mathcal{P}$ could possibly be performed (e.g., the semantics defined in [24] involves the application of a minimality criterion to filter "candidate" world views and reject those $\mathcal{W}'$ that make "larger assumptions" on known atoms –see [24] for a formal description). If the resulting epistemic interpretation coincides with the initial set $\mathcal{W}$, then it is an $\mathcal{S}$-world view for Π. This process has to be completed for all the epistemic interpretations. Hence, being At_Π the set of the atoms occurring in (heads of) rules of Π, one has to find all the subsets of At_Π, and all the sets of such subsets. Note that some simplifications are possible, for example, one can ignore those epistemic interpretations that contain two sets, one included in the other.

A "quickly prototyped" solver for given $\mathcal{S}$, $\mathcal{R}$, and $\mathcal{P}$, returning all world views of any program Π, is obtained by running on Π the pipeline of the following modules:

1. A module $M_{\mathcal{W}}$ that computes all epistemic interpretations $\mathcal{W}_1$, ..., $\mathcal{W}_k$ for Π (i.e., all sets of subsets of At_Π);
2. A module M_{red} that applies, for each $\mathcal{W}_i$, the reduct $\mathcal{R}$ to Π, and generates the reduct program Π_i (which is an ASP program);
3. A module M_{ASP} that, for each $i = 1, \ldots, k$, computes the set of answer sets of Π_i, let SMs_i denote such set;
4. In case a post-processing $\mathcal{P}$ is required, a module $M_{\mathcal{P}}$ applying $\mathcal{P}$ to select the desired candidate SMs_i's;
5. A module M_{chk} that checks each SMs_i produced by the previous step and selects those which are world views w.r.t. $\mathcal{S}$, as they coincide with the corresponding $\mathcal{W}_i$.

Clearly, the correctness of this solver w.r.t. $\mathcal{S}$ depends upon the correct implementation of the various modules, that, however, should not be difficult to ensure. In fact, each module copes with a single aspect and will thus be sufficiently transparent and reasonable in size.

Let us consider, as an example, the simple ELP program

$$a \leftarrow b.$$
$$b \leftarrow not\ \mathbf{K}\ not\ a.$$

We have that $At_\Pi = \{a, b\}$ and the possible epistemic interpretations $\mathcal{W}_i$ are subsets of $\big\{\emptyset, \{a\}, \{b\}, \{a, b\}\big\}$. Consider for instance $\mathcal{W}_1 = [\{a, b\}]$. Clearly, both $\mathcal{W} \models a$ and $\mathcal{W} \models b$ hold. Choosing the

semantics $\mathcal{S}$ and the notion of reduct stated by Def. 1, the G94-reduct Π_1 w.r.t. $\mathcal{W}_1$ obtained by module M_{red} is:

$$a \leftarrow b.$$
$$b \leftarrow not \bot. \qquad \text{(i.e., the fact } b\text{)}$$

for which module M_{ASP} computes $SMs_1 = [\{a, b\}]$. No post-processing is needed and module M_{chk} validates $\mathcal{W}_1 = SMs_1$ as a world view of the ELP program. Similarly, the pipeline obtains $[\emptyset]$ as the only other world view of the given ELP program.

6 Fast prototyping of a solver: the Implementation

We implemented the pipeline of modules described in Section 5 as an ASP Chef recipe, which we illustrate below. The complete recipe is accessible, usable, and modifiable through the ASP Chef web app, reachable by the following clickable link:

ELP in ASP Chef .

Note that the ASP Chef recipe is directly encoded in the above clickable link. The link itself contains the full specification of the recipe in an encoded form.[2] When the link is opened in a web browser, the ASP Chef app is fetched from https://asp-chef.alviano.net/ and run locally on the reader's machine, decoding and interpreting the recipe from the link itself. No user data is transferred to the ASP Chef site. The fact that the complete recipe is encoded within the link ensures full transparency and reproducibility. Once the recipe/link is accessed by the web browser, the user can freely modify the recipe. In turn, this generates a different link that encodes the updated recipe.

Let us remark that this recipe is not optimal in the sense that we chose to implement the pipeline for expository purposes rather than aiming for an optimized implementation. Also, we chose to implement each step of the pipeline separately, not necessarily using a minimal number of ingredients. Hence, a more compact recipe could be obtained, for instance, by merging different ingredients or by skipping some steps that perform some inessential processing (such as filtering of atoms that are useless for the subsequent ingredients to reduce the size of `clingo-wasm` input).

Figure 1 shows the ASP Chef web app and the initial part of the recipe. Let us describe the main crucial parts of such a recipe. As can be seen from Figure 1, the ingredients in the recipe are numbered (see also the complete recipe obtainable through the link reported earlier). In what follows we will refer to such numbering.

Regarding the input format, since ASP Chef ingredients expect to be fed with a sequence of sets of atoms, we encoded the ELP program Π as a set of ASP atoms. Namely, to encode a rule of the form (1) we used the fact

`rule(head(`A_1`,...,`A_g`), body(`L_1`,...,`L_n`)).`

A subjective literal $\mathbf{K}\,G$ is encoded by the term `k(`G`)`, while negation is represented by the functor `neg`. So, we reserved the symbols `neg` and `k` and they cannot occur in input. For example, the program (2) of Section 3 is encoded as the set of four facts:

```
rule(head(a), body(neg(b))).
rule(head(b), body(neg(a))).
rule(head(e), body(neg(k(f)))).
rule(head(f), body(neg(k(e)))).
```

These facts are the single set of facts in the input sequence for the recipe (cf., the input panel in the top right corner of Figure 1). The list of ingredients composing the recipe is shown in the central panel of the web app. Utilizing an INTROSPECTION TERMS ingredient (ingredient `#1` in the recipe), one introduces some Lua functionalities (see [1]) useful to decompose input facts in order to access their subterms. For instance, the first ingredient in Figure 1 introduces the functions `@functor`, `@arity`, and `@argument`, later used to access the functor, the arity, and the arguments of a composed term, resp.[3] Consequently, the following ASP program can process the input set of facts to detect the atoms and the literals, both objective and subjective, that occur in the heads and bodies of the input ELP rules. This ASP program is evaluated the second ingredient, which is a SEARCH MODELS ingredient.

```
rule_head(rule(H,B), @argument(H,I)) :-
    rule(H,B), I = 1..@arity(H).
rule_body(rule(H,B), @argument(B,I)) :-
    rule(H,B), I = 1..@arity(B).
% literals occurring in heads or in bodies:
hlit(L) :- rule_head(_,L).
blit(L) :- rule_body(_,L).
% literals/atoms occurring in bodies:
atom(L) :- hlit(L).
atom(A) :- blit(neg(A)),
    @functor(A)!="neg", @functor(A)!="k".
atom(A) :- blit(k(A)), @functor(A)!="neg",
    @functor(A)!="k".
blit(L) :- blit(neg(L)).
blit(L) :- blit(k(L)).
% literals/atoms in subj. literals:
klit(A) :- blit(k(A)).
klit(L) :- klit(neg(L)).
```

This ingredient simply executes the ASP program joined with the input facts and generates the corresponding stable models. In this case, thanks to the mentioned Lua functions, the ASP program collects all literals/atoms occurring in heads and bodies of ELP rules (represented via the predicates `hlit/1` and `blit/1`, resp.), as well as those occurring in subjective literals (`klit/1`). Moreover, in the output of this ingredient, one obtains the set At_Π encoded as a collection of facts of the form `atom(A)`. At this point, another SEARCH MODELS ingredient (ingredient `#4`) evaluates such collection together with the simple ASP program:

```
{guess_true(A)} :- atom(A), hlit(A).
```

to generate a sequence of answer sets, each of them containing a possible subset of At_Π. Then, a SELECT PREDICATES ingredient (`#5`) filters relevant predicates and a MERGE ingredient (`#6`) combines the computed subsets of At_Π in a single set by distinguishing/indexing their elements by the predicate `__atomset__` (i.e., each atom $\langle atom \rangle$ occurring in the i-th subset is encoded in output by an atom of the form `__atomset__(`i`,`$\langle atom \rangle$`)`). Now the following ASP program are used by three SEARCH MODELS ingredients (numbered `#7`, `#9`, and `#10`, resp.) to process such "indexed atoms".

```
numset(SetID) :- __atomset__(SetID,_).

{selectset(SetID)} :- numset(SetID).
```

[2] This is why the link appears as a long, unreadable, string. To facilitate the reader of the printed version of this paper, we made the link accessible also indirectly via https://tinyurl.com/ELPASPChefG94.

[3] Lua is a high-level, multi-paradigm, lightweight, cross-platform programming language that is primarily intended for embedded applications. It is frequently referred to as a "multi-paradigm" language since it offers a set of general features that can be extended as needed to accommodate various problem kinds [18].

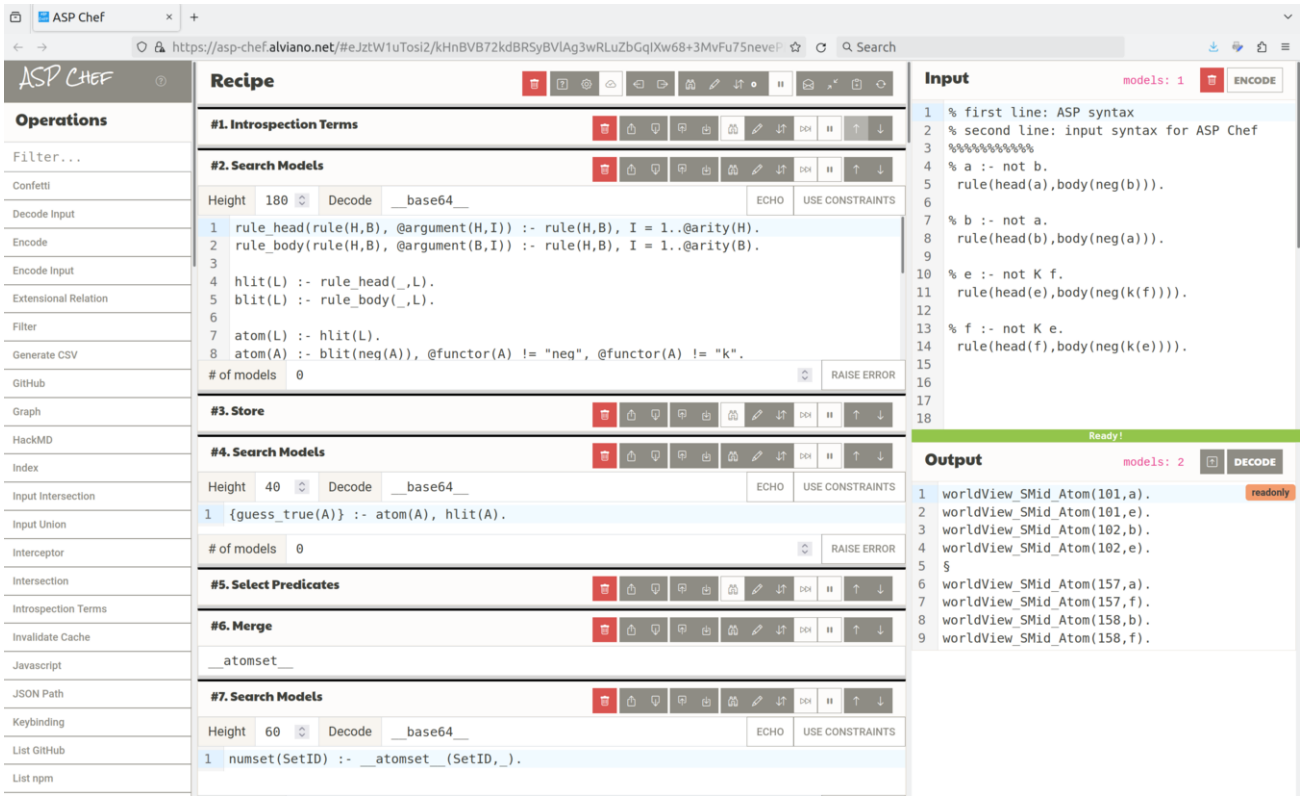

Figure 1. The ASP Chef recipe to compute the world views of an ELP program under the semantics G94.

```
% count the number of selected IDs
numOfSetsInW(Count) :-
    Count==#count{I:selectset(I)}.
setInW(SetID,A) :- __atomset__(SetID,A),
    selectset(SetID).
```

The first rule simply collects all sets indices `SetID` introduced by the previous ingredient. The second rule generates all possible selections of such indices/sets (i.e., the possible epistemic interpretations).

The computation time of this step can increase significantly, depending on the size of the input program. This is because, in principle, all subsets of the power set of At could be considered as candidate epistemic interpretations. Hence, while in general the number of rules in the input program is not a significant limiting factor, the cardinality of At can have a large impact on the time and amount of memory required by ASP Chef to complete this step. To improve the usability of the tool, we introduced a SET TIMEOUT ingredient (`#8`) that can be used to set a timeout for the execution of ingredient (`#9`). Furthermore, it is also possible to control the generation of the candidate epistemic interpretations by imposing a constraint on their cardinality. This can be done by commenting out the second of the above rules and activating these rules (currently commented out in ingredient (`#9`)):

```
B1 {selectset(SetID) : numset(SetID)} B2 :-
    cardOfEpInterp(B1,B2).
cardOfEpInterp(1,2).  % set bounds
```

An additional code fragment in ingredient `#9` avoids generating candidate solutions containing two sets one subset of the other:

```
notsub(S,R) :- selectset(S), selectset(R),
    S!=R, __atomset__(S,guess_true(A)),
    not __atomset__(R,guess_true(B)).
:- not notsub(S1,S2), selectset(S1),
    selectset(S2), S1!=S2.
```

Each of the generated epistemic interpretations $\mathcal{W}_i$ appears in one answer set of the ingredient output, and its members (the candidate stable models of Π_i) are encoded by a list of facts of the form `setInW(SetID,A)`. This completes step 1 in the pipeline.

A further SEARCH MODELS (ingredient `#14`, in the recipe) evaluates the following program for each $\mathcal{W}_i$:

```
modeledByW(A) :- atom(A), numOfSetsInW(N),
    N>0, N==#count{I,A :
    setInW(I,guess_true(A)), selectset(I)}.

modeledByW(neg(A)) :- atom(A),
    0==#count{I,A :
    setInW(I,guess_true(A)), selectset(I)}.
```

and infers which literals $\mathcal{W}_i$ models (by simply counting the modelling sets/IDs and comparing their number with the cardinality of $\mathcal{W}_i$. At this point the reduct of Π can be computed, w.r.t. each of the epistemic interpretation $\mathcal{W}_i$. In the case of semantics G94, this can be achieved by a SEARCH MODELS (ingredient `#18`) which processes the following ASP rules (for each of the $\mathcal{W}_i$s):

```
red_blit(k(L),true) :- blit(k(L)),
    modeledByW(L).
red_blit(k(L),false) :- blit(k(L)),
    not modeledByW(L).
red_blit(neg(L),neg(L)) :- blit(neg(L)),
    @functor(L)!="k".
red_blit(L,L) :- blit(L),
    @functor(L)!="neg", @functor(L)!="k".
red_blit(neg(k(L)),false) :-
    blit(neg(k(L))), modeledByW(L).
```

```
red_blit(neg(k(L)),true) :-
    blit(neg(k(L))), not modeledByW(L).
% reduced rules head and body literals:
red_rule_head(rule(H,B), @argument(H,I)) :-
    rule(H,B), I = 1..@arity(H).
red_rule_body(rule(H,B), R) :-
    rule_body(rule(H,B), L), red_blit(L,R).
```

The first six rules assign to each literal in the input program Π a "substitute", according to Def. 1. Such substitute may be the literal itself or one of the truth values `false` and `true` (denoting the values $\bot$ and $\top$ used in Def. 1). Then, depending on which subjective literals are modeled by the epistemic interpretation $\mathcal{W}_i$, facts of the form `red_rule_head(`$\langle rule \rangle$`,`$\langle lit \rangle$`)` and `red_rule_body(`$\langle rule \rangle$`,`$\langle lit \rangle$`)` are derived, representing the rules of the reduced program. Each reduced program is represented in a different set of atoms in the output sequence of the ingredient. This completes step 2 in the pipeline of Section 5.

All the sets so obtained are then processed by another SEARCH MODELS (ingredient `#20`) evaluating the program

```
% detect falsified reduced rules bodies:
red_body_false(R) :- red_rule_body(R,false).
% infer true literals w.r.t. reduced rules:
true(L) : red_rule_head(rule(H,B), L) :-
  rule(H,B), not red_body_false(rule(H,B));
  true(N):red_rule_body(rule(H,B),N),
      @functor(N) !="neg",
      @functor(N) !="true";
  not true(M):red_rule_body(rule(H,B),neg(M)),
      @functor(M) !="false".
```

to obtain its answer sets (encoded, one in each set of the output sequence, by a collection of facts `true(`$\langle atom \rangle$`)`. At this point, each element of each SMs_i (for all i) is encoded in a distinct set of atoms in the output sequence of the ingredient. Few ingredients are used to gather in a single set the collection SMs_i, for each i. This operation outputs the sequence of SMs_is and completes step 3 in the pipeline of Section 5.

Finally, step 5 of the pipeline is performed (notice that step 4 is not needed for the semantics G94). For brevity, we omit the details: the remaining part of the recipe manipulates the collection SMs_i of answer sets of each reduced program and compares it with $\mathcal{W}_i$, filtering out those that do not match. The output is then processed for better readability, and the world views of Π are listed by facts of the form `worldView_SMid_Atom(`$\langle SMid \rangle$`,`$\langle atom \rangle$`)`. For instance, this is the output for the ELP program (2):

```
worldView_SMid_Atom(101,a).
worldView_SMid_Atom(101,e).
worldView_SMid_Atom(102,b).
worldView_SMid_Atom(102,e).
§
worldView_SMid_Atom(157,a).
worldView_SMid_Atom(157,f).
worldView_SMid_Atom(158,b).
worldView_SMid_Atom(158,f).
```

The two sets in the output sequence (separated by §) represent two different world views, each composed of two answer sets (identified by different numeric IDs), each composed of two atoms.

We conclude this section by observing that the pipeline described in Section 5 and implemented in ASP Chef can be easily exploited to mechanize any reduct-based semantics for ELPs (hence the "fast prototyping"). It simply suffices to modify a single ingredient. Namely, ingredient `#18` that implements the module M_{red} of the pipeline and, for each $\mathcal{W}_i$, computes the reduct program Π_i. This is equivalent to providing adequate alternative definitions of the predicates `red_blit/2` and `red_rule_body/2` seen before, according to the specific notion of reduct at hand. The rest of the recipe remains unchanged.

Consider for instance the semantics G11 introduced in [15]:

Definition 2 (G11-world views)**.** *The G11-reduct of* Π *w.r.t. a non-empty set of interpretations* W *is obtained by: (i) replacing by* $\bot$ *every subjective literal* $L \in Body_{subj}(r)$ *such that* $W \not\models L$*; (ii) removing all other occurrences of subjective literals of the form* $not\ \mathbf{K}L$*; (iii) replacing all other subjective literals of the form* $\mathbf{K}L$ *by* L*.*

The set W *is a G11-world view of* Π *iff* W *coincides with the set of all stable models of the G11-reduct of* Π *w.r.t.* W*.*

The following is a possible encoding of the G11-reduct which involves a change in the definition of `red_blit/2` only.

```
red_blit(k(L),false) :- blit(k(L)), not
    modeledByW(L).
red_blit(neg(k(L)),false) :-
    blit(neg(k(L))), modeledByW(L).
red_blit(neg(k(L)),true) :-
    blit(neg(k(L))),
    not modeledByW(@argument(@argument(L,1),1)).
red_blit(k(L),L) :- blit(k(L)),
    modeledByW(L).
red_blit(neg(L),neg(L)) :- blit(neg(L)),
    @functor(L) !="k".
red_blit(L,L) :- blit(L),
    @functor(L) !="neg", @functor(L) !="k".
```

We conclude this section by observing that the approach can be adapted to deal with other epistemic operators such as **M** and **not**, as they can be syntactically reduced to **K**. The easiest way to achieve this would be to add an initial ingredient (actually, a SEARCH MODELS) that processes the input ELP rules to preliminarily translate the occurrences of **M** and **not** operators and replace them with their **K**-based formulations.

7 Conclusions

The quest for the "right" semantics of Epistemic Logic Programs continues and can be facilitated by the possibility of experimenting with new semantic approaches, going beyond the tiny programs that one may consider on paper. To allow for such experimentations, we devised a method for fast prototyping of solvers for reduct-based semantic approaches. We illustrated the method for the G94 and the G11 semantics (which implies that the method is also indirectly applicable to FAAEL), and we showed that it is, however, easily customizable to every other reduct-based approach. This is due to the modular definition and the corresponding modular implementation in ASP Chef. As a matter of fact, we verified through the proposed tool all the examples occurring in the literature plus others. A current limitation is the limited scalability of the implementation, which still does not allow the application of the method to larger programs. Nonetheless, we emphasize that the aim of this work is not to compete with optimized solvers, but to enable researchers to perform simple experiments (not only manually) while developing new ELP semantics, with limited implementation effort.

Thus, future work will be concerned with implementing and experimenting with other semantics and improving the efficiency of the implementation. We will also investigate the possibility of extending the method to semantic approaches that are not reduct-based.

References

[1] M. Alviano, D. Cirimele, and L. A. Rodriguez Reiners. Introducing ASP recipes and ASP Chef. In *Proceedings of the ICLP'23 Workshops*, volume 3437 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023.

[2] G. Brewka, T. Eiter, and M. T. (eds.). Answer set programming: Special issue. *AI Magazine*, 37(3), 2016.

[3] P. Cabalar, J. Fandinno, and L. Fariñas del Cerro. Autoepistemic answer set programming. *Artif. Intell.*, 289:103382, 2020.

[4] P. Cabalar, J. Fandinno, and L. Fariñas del Cerro. Splitting epistemic logic programs. *Theory Pract. Log. Program.*, 21(3):296–316, 2021.

[5] S. Costantini. About epistemic negation and world views in epistemic logic programs. *Theory Pract. Log. Program.*, 19(5-6):790–807, 2019. doi: 10.1017/S147106841900019X.

[6] S. Costantini and A. Formisano. Negation as a resource: a novel view on answer set semantics. *Fundam. Informaticae*, 140(3-4):279–305, 2015. doi: 10.3233/FI-2015-1255.

[7] S. Costantini and A. Formisano. Epistemic logic programs: A study of some properties. *Theory Pract. Log. Program.*, 24(3):482–504, 2024. doi: 10.1017/S1471068424000012.

[8] A. Dovier, A. Formisano, and E. Pontelli. An experimental comparison of constraint logic programming and answer set programming. In *Proc. of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, Canada*, pages 1622–1625. AAAI Press, 2007.

[9] A. Dovier, A. Formisano, and E. Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. of Experimental and Theoretical Artificial Intelligence*, 21(2):79–121, 2009. doi: 10.1080/09528130701538174.

[10] A. Dovier, A. Formisano, and E. Pontelli. Parallel answer set programming. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 237–282. Springer, 2018. doi: 10.1007/978-3-319-63516-3_7.

[11] E. Erdem, M. Gelfond, and N. Leone. Applications of answer set programming. *AI Mag.*, 37(3):53–68, 2016.

[12] L. Fariñas del Cerro, A. Herzig, and E. I. Su. Epistemic equilibrium logic. In Q. Yang and M. J. Wooldridge, editors, *Proc. of IJCAI 2015*, pages 2964–2970. AAAI Press, 2015.

[13] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool, 2012. ISBN 978-3031004339.

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

[15] M. Gelfond. New semantics for epistemic specifications. In J. P. Delgrande and W. Faber, editors, *Proc. of LPNMR'11*, volume 6645 of *LNCS*, pages 260–265. Springer, 2011.

[16] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. of the 5th Intl. Conf. and Symp. on Logic Prog.*, pages 1070–1080. MIT Press, 1988.

[17] M. Gelfond and H. Przymusinska. Definitions in epistemic specifications. In A. Nerode, V. W. Marek, and V. S. Subrahmanian, editors, *Proc. of the 1st Intl. Workshop on Logic Programming and Nonmonotonic Reasoning*, pages 245–259. The MIT Press, 1991.

[18] R. Ierusalimschy. *Programming in Lua*. Lua.org, 4 edition, 2016. ISBN 978-8590379867.

[19] P. T. Kahl and A. P. Leclerc. Epistemic logic programs with world view constraints. In A. D. Palù, P. Tarau, N. Saeedloei, and P. Fodor, editors, *Tech. Comm. of ICLP 2018*, volume 64 of *OASIcs*, pages 1:1–1:17. Schloss Dagstuhl, 2018.

[20] B. Kaufmann, N. Leone, S. Perri, and T. Schaub. Grounding and solving in answer set programming. *AI Mag.*, 37(3):25–32, 2016. doi: 10.1609/AIMAG.V37I3.2672.

[21] A. P. Leclerc and P. T. Kahl. A survey of advances in epistemic logic program solvers. In J. Fandinno and J. Fichte, editors, *Proc. of the 11th Workshop on Answer Set Programming and Other Computing Paradigms ASPOCP'18*, 2018. URL http://arxiv.org/abs/1809.07141.

[22] V. Lifschitz. Thirteen definitions of a stable model. In A. Blass, N. Dershowitz, and W. Reisig, editors, *Fields of Logic and Computation*, volume 6300 of *LNCS*, pages 488–503. Springer, 2010.

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

[24] Y. Shen and T. Eiter. Evaluating epistemic negation in answer set programming. *Artificial Intelligence*, 237:115–135, 2016.

[25] E. I. Su. Epistemic answer set programming. In F. Calimeri, N. Leone, and M. Manna, editors, *Proc. of JELIA'19*, volume 11468 of *LNCS*, pages 608–626. Springer, 2019.

[26] M. Truszczynski. Revisiting epistemic specifications. In M. Balduccini and T. C. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *LNCS*, pages 315–333. Springer, 2011.

[27] Web references of ASP solvers, 2025. Clingo: potassco.org; Cmodels: www.cs.utexas.edu/users/tag/cmodels; DLV: www.dlvsystem.it; WASP: alviano.github.io/wasp.