Embedded Implications and Minimality in ASP

Mauricio Osorio and Magdalena Ortiz

Universidad de las Américas, CENTIA
Sta. Catarina Mártir, Cholula, Puebla
72820 México
{josorio,is103378}@mail.udlap.mx

Abstract. Disjunctive logic programs under the answer sets semantics play a very significant role in knowledge representation and non monotonic reasoning. As the semantics has been extended to wider classes of programs, it has been observed that implication in the body of rules can be interesting, both for theoretical and practical issues. Here we present an extension of the answer sets semantics to programs with embedded implications in the body of the rules. Since the answer sets of these extended programs are not necessarily minimal, we introduce the notion of *rigid* programs as a condition that assures the minimality of answer sets for arbitrary theories. We also introduce an extended family of logic programs with restricted use of implication in the body of the rules and address some practical knowledge representation issues through this extension.

1 Introduction

Disjunctive logic programming under the stable model semantics, also known as Answer Sets Programming (ASP) is an useful and expressive formalism for knowledge representation and reasoning [1]. As the popularity of this formalism has increased, many researchers have tried to use it for representing and solving a wide range of problems. With these developments new challenges for the ASP community have been met and many extensions to the syntax and semantics of disjunctive logic programs have been proposed [9,10,12].

Many research efforts in this area have focused on extending the semantics to wider classes of programs. The use of implication in the body of rules has been recognized lately as an extension that allows more natural problem solving, and the need of it arises both while modeling real applications as well as for research purposes. In [12] the authors discuss that for some knowledge representation problems, an extension to the answer set semantics is needed. They introduce *parametric connectives* and present the solution to some problems using this extension. Intuitively, the reading of parametric conjunctions corresponds to a quantified implication, and it can be seen that many of the problems the authors present could also be represented in an uniform and natural way with the extension we propose. The importance of this extension is also clear in the remark done by Michael Gelfond via e-mail communication: *"The ability to use implication in the body seems to suggest the following translation: 'r is true if every element with property p has property q' The natural translation is:* $\forall(X)(p(X) \rightarrow q(X)) \rightarrow r$*. If no implication is allowed in the formal language the translation of this English statement loses its universal character. It now depends on the context and is prone to error."*

In this paper we extend the answer set semantics to programs with embedded implications in the body of rules. We discuss some of the main issues related to this extension, namely minimality and translations to simpler classes of programs. We will focus on the aspects that we consider of particular relevance when modeling applications and solving problems using ASP. Our main result is the introduction of *rigid* programs. *Rigidity* is a sufficient condition to assure the minimality of the answer sets of any given theory. The relationship between classes of programs and minimal models is studied according to the proposed rigidity condition. We also introduce an extended family of programs that allows the use of implication in the body of the rules assuring minimality of answer sets and is at the same time useful for modeling some interesting problems.

2 Background

2.1 Propositional Logic

The language of propositional logic has an alphabet consisting of propositional symbols: $p_0, p_1, \ldots$; connectives: $\wedge, \vee, \leftarrow, \bot$ and auxiliary symbols: $(,)$. Propositional symbols are also called *atoms* or *atomic propositions*. Formulas and theories are defined as usual in logic. The formula $\neg F$ is introduced as an abbreviation of $\bot \leftarrow F$, and the formula $F \rightarrow G$ is just another way of writing the formula $G \leftarrow F$. $F \equiv G$ is an abbreviation of $(G \rightarrow F) \wedge (F \rightarrow G)$. A signature $\mathcal{L}$ is a finite set of propositional symbols. The *signature* of F, denoted as $\mathcal{L}_F$, is the set of propositional symbols that occur in F. A *literal* is either an atom a (a positive literal) or the negation of an atom $\neg a$ (a negative literal). A negated literal is the negation sign $\neg$ followed by any literal, i.e. $\neg a$ or $\neg\neg a$.

2.2 Logic Programs

A clause is a formula of the form $H \leftarrow B$ where H and B, arbitrary formulas in principle, are called the *head* and *body* of the clause respectively. If $H = \bot$ the clause is called a *constraint* and can be written as $\leftarrow B$. Analogously, if $B = \top$ then the clause is called *fact* and can be written just by H. A *general clause* is a clause where the head is a (possibly empty) disjunction of atoms and the body a conjunction of literals. A *logic program* is then a finite set of clauses. A logic program is also a theory, we will use the terms as synonyms. A set of general clauses is a *general program*. Let P be a program and M a set of atoms such that $M \subseteq \mathcal{L}_P$ then we define $\widetilde{M} = \mathcal{L}_P \setminus M$.

2.3 General Definitions

Some general concepts can be defined on any logic. Here we will use I to refer to *intuitionistic* logic. For any given logic X, and for given set of atoms M and a program P we will write

$P \vdash_{\mathrm{X}} M$ to abbreviate $P \vdash_{\mathrm{X}} a$ for all $a \in M$ and $P \nvdash_{\mathrm{X}} M$ to denote the fact that P *is consistent (w.r.t. logic* X*) and* $P \vdash_{\mathrm{X}} M$.

The set $\mathbf{P}$ of *positive formulas* is the smallest set containing all formulas without negation connectives ($\neg$) [1]. For a given set of formulas Γ, the *positive subset* of Γ, denoted as $\mathrm{Pos}(\Gamma)$, is the set $\Gamma \cap \mathbf{P}$.

Lemma 1. *Let* Γ *be a subset of* $\mathbf{P} \cup \mathbf{N2}$*, and let* $A \in \mathbf{P}$ *be a positive formula. If* $\Gamma \vdash_{\mathrm{I}} A$ *then* $\mathrm{Pos}(\Gamma) \vdash_{\mathrm{I}} A$.

Lemma 2. *Let* T *be any theory and* L *a set of negative literals such that* $\mathcal{L}_T \cap \mathcal{L}_L = \emptyset$*. For any formula* A *such that* $\mathcal{L}_A \cap \mathcal{L}_L = \emptyset$ *if* $T \cup L \vdash_{\mathrm{I}} A$ *then* $T \vdash_{\mathrm{I}} A$.

2.4 Answer sets

We now define the basic background for Answer sets (or equivalently stable models). Before we go on into the stable models semantics, we will point out that in the logic programs presented we use two types of negation. The negation *not* is the usually called *default negation* and is the logic programming counterpart of the logical negation $\neg$, which we have been using. The other negation $\sim$ represents the *explicit* or *true* negation. We use it for practical purposes, but it does not affect the semantics. Any program with this negation can be easily translated into one without by simply renaming atoms and adding constraints.

Stable models were first defined by Gelfond and Lifschitz [3] and they are today the most accepted semantics for disjunctive logic programs. Their logical characterization and extensions to wider theories is due to [5,8,10,11], among others. The material here presented is taken from [9]. It provides a characterization of answer sets and minimal models in terms of intuitionistic logic (the result holds for any intermediate logic). The authors present the results augmented programs. We now extend the definition to any arbitrary theory.

Definition 1 (Answer set of a program). *Let* P *be any theory and* M *a set of atoms.* M *is called an answer set for* P *iff* $P \cup \neg\widetilde{M} \cup \neg\neg M \nvdash_{\mathrm{I}} M$

We will use the term *minimal model* as usual in logic programming [4], i. e., the set M is a minimal model of P if M in a model of P and it is minimal (w.r.t set inclusion) among all other models of P.

Theorem 1. *Let* P *be any theory and* M *a set of atoms.* M *is a minimal model and an answer set of* P *iff* $P \cup \neg\widetilde{M} \nvdash_{\mathrm{I}} M$.

We say that two programs P_1 and P_2 are equivalent under the answer set semantics, written as $P_1 \equiv_{\mathrm{stable}} P_2$, if they have exactly the same answer sets.

[1] The formula consisting of $\bot$ alone is also considered a positive formula.

3 Embedded implications, minimality and answer sets

One of the key issues when using embedded implications is that the answer sets of this kind of programs are no longer necessarily minimal. Minimality plays an important role in models of logic programs. In the following section, we address it from a formal perspective. Our main result is the definition of a simple sufficiency condition that assures minimality of answer sets. We also analyze some classes of programs with respect to the given condition. Our result is based on the following principle: An arbitrary theory can be reduced with respect to a set of negative literals into a program that does not contain the given literals but still preserves the proving power of the first one. If this new program can be translated into a positive one without gaining proof power, then the program holds our *rigidity* condition, which assures that the minimality of answer sets is preserved.

3.1 Reducing a theory

In this section, we present some reductions that can be applied to programs. We will later use these reductions as a theoretical tool to define our main result, the *rigidity* condition. The reductions here presented are similar to some known transformations, like those in [9], but here they can be applied to any arbitrary theory.

Definition 2 ($\bot$ reduction). *For a given theory P, we define its reduction $redu\bot(P) := \{redu\bot_f(\alpha) : \alpha \in P\}$. Reduction $redu\bot_f(\alpha)$ over formulas is obtained by removing $\bot$ or $\top$ from formulas with rules like [2]: replace $\alpha \vee \bot$ or $\bot \vee \alpha$ by α; replace $\alpha \vee \top$ or $\top \vee \alpha$ by $\top$; replace $(((\alpha \to \bot) \to \bot) \to \bot)$ by $\alpha \to \bot$; etc. until no more reductions can be applied.*

Definition 3 (Negative reduction). *Let α be a formula and L_p a set negative literals, then $redu1_f(\alpha, L_p)$ is obtained replacing every occurrence of an atom a in α by $\bot$ if $\neg a \in L_p$. Let P be a theory and L_p a set of negative literals. We define $redu1(P, L_p) := redu\bot(P', L_p)$, where $P' := \{redu1_f(\alpha, L_p) | \alpha \in P\}$.*

Definition 4 (Negative2 reduction [7]). *Let L_p be a set of negated negative literals. The reduction $redu2_f$ is defined over formulas recursively as follows:*

1. *for an atom a, $redu2_f(a, L_p) = a$.*
2. *for a formula $\alpha \to \bot$, $redu2_f(\alpha \to \bot, L_p)$ is the formula obtained by replacing every occurrence of an atom a in $\alpha \to \bot$ by $\top$ if $\neg\neg a \in L_p$.*
3. *For a formula $\alpha \# \beta$, where $\# \in \{\vee, \wedge, \to\}$, $redu2_f(\alpha \# \beta, L_p) = redu2_f(\alpha) \# redu2_f(\beta)$. We assume that if $\#$ is $\to$ then β is not $\bot$.*

[2] In a slight abuse of notation, we will use the $\top$ symbol, which is an abbreviation of $\bot \leftarrow \bot$, as an explicit constant.

Let P be a theory, L_p a set of negated negative literals and $P' := redu_{\perp}\{(redu2_f(\alpha, L_p)|\alpha \in P\}$ We define $redu2(P, L_p) := P' \setminus \{\alpha \in P' : \vdash_{\mathrm{I}} \alpha\}$. Finally, $redu2^*(P, L_p)$ is obtained by applying $redu2(P, L_p)$ until no more reductions can be obtained.

The following propositions refer to some properties of the programs obtained after applying the reductions [3].

Proposition 1. *Let P be a theory and L_p a set of negative literals, then $redu1(P, L_p) \cup L_p \equiv_{\mathrm{I}} P \cup L_p$ and $\mathcal{L}_{L_p} \cap \mathcal{L}_{redu1(P,L_p)} = \emptyset$.*

Proposition 2. *Let P be a theory and L_p a set of negated negative literals, then we have $redu2^*(P, L_p) \cup L_p \equiv_{\mathrm{I}} P \cup L_p$.*

Remark 1. Let P be a theory and let M be a set of literals such that $M \subseteq \mathcal{L}_P$. Then $redu2^*(redu1(P, \neg\widetilde{M}), \neg\neg M) \subseteq \mathbf{P}$.

3.2 Rigid programs

Preserving minimality. Now that we have given a suitable background, we present our main result: the *rigidity* condition. This is a property of programs which assures the minimality of answer sets of arbitrary theories. We analyze this condition with respect to some well known classes of programs and we introduce a class of programs with embedded implications in the body of the rules which also satisfies the rigidity condition.

Proposition 3. *Let P be any theory and $M \subseteq \mathcal{L}_P$. If $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$, then we have that $redu2^*(redu1(P, \neg\widetilde{M}), \neg\neg M) \Vdash_{\mathrm{I}} M$.*

Proof. Let $P_1 := redu1(P, \neg\widetilde{M})$ and $P_2 := redu2^*(redu1(P, \neg\widetilde{M}), \neg\neg M)$. Since $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$, and also $redu1(P, L_p) \cup L_p \equiv_{\mathrm{I}} P \cup L_p$ (Proposition 1) we have that $P_1 \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$. Analogously by Proposition 2 we show $P_2 \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$. It follows by Lemma 2 that $P_2 \cup \neg\neg M \Vdash_{\mathrm{I}} M$ and since M is positive, Lemma 1 proves that $P_2 \Vdash_{\mathrm{I}} M$.

Definition 5 (Rigid theory). *Let P be a theory. We say that P is rigid if for every M such that $M \subseteq \mathcal{L}_P$ we have that $redu1(P, \neg\widetilde{M}) \vdash_{\mathrm{I}} redu2^*(redu1(P, \neg\widetilde{M}), \neg\neg M)$.*

[3] For the full proofs refer to http://mailweb.udlap.mx/∼is103378/research/rigid

Theorem 2. *Suppose P is a rigid theory and $M \subseteq \mathcal{L}_P$. Then $P \cup \neg\widetilde{M} \Vdash_{\mathrm{I}} M$ iff $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$.*

Proof. First suppose $P \cup \neg\widetilde{M} \Vdash_{\mathrm{I}} M$. Since $A \rightarrow \neg\neg A$ is an intuitionistic theorem $P \cup \neg\widetilde{M} \Vdash_{\mathrm{I}} \neg\neg M$. Therefore $P \cup \neg\widetilde{M} \cup \neg\neg M$ is consistent and since intuitionistic logic is monotonic we have $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$. Now suppose $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$, its immediate that $P \cup \neg\widetilde{M}$ is consistent. Now, we show that $P \cup \neg\widetilde{M} \vdash_{\mathrm{I}} M$. Let $P' := redu1(P, \neg\widetilde{M})$ and $P'' := redu2^*(P', \neg\neg M)$. By Proposition 3 we know that $P'' \vdash_{\mathrm{I}} M$. Since P is rigid, $P' \vdash_{\mathrm{I}} P''$ and we get $P' \vdash_{\mathrm{I}} M$. By $P \cup \neg\widetilde{M} \equiv_{\mathrm{I}} P' \cup \neg\widetilde{M}$ (Proposition 1), we get$P \cup \neg\widetilde{M} \vdash_{\mathrm{I}} M$ as desired.

Proposition 4. *Let P be a rigid program, then M is an answer set of P implies that M is a minimal model of P.*

Corollary 1. *Let P be a rigid program. M is an answer set of P if and only if $P \cup \neg\widetilde{M} \Vdash_{\mathrm{I}} M$.*

4 Classes of Rigid Programs

In this section, we present two useful classes of logic programs that hold the rigidity condition, hence their answer sets are always minimal. The first one is the well known class of general programs. The second one is a family of programs that supports implications in the body of rules in a restricted way.

4.1 General Programs

The first family of logic programs that we prove to be rigid are general programs. This is an useful result, since general programs are the most widely used disjunctive programs, and most software implementations that compute stable models support them.

Proposition 5. *Every general program is rigid.* [4]

Corollary 2. *Let P be a general program, then M is a answer set of P implies that M is a minimal model of P.*

4.2 Implication-Embedded Disjunctive Logic Programs

We now present a class of programs that extends general ones allowing the use of implications in the body of the rules. Despite its syntactical restrictions, the class is suitable for solving some knowledge representation problems. We will also analyze the rigidity condition of this kind of programs.

[4] The proof is based on the fact that applying $redu2$ to the $redu1$ of the program will only remove clauses.

Definition 6 (Positive embedded program). *A* positive embedded conjunct *is either a literal or a formula of the form* $(a \rightarrow b)$*, where* a *and* b *are atoms. A* positive embedded clause *is a clause of the form* $H \leftarrow B$*, where* H *is an atom and* B *is a conjunction of positive embedded conjuncts.* P *is a* positive embedded program *if for every clause* $\alpha \in P$*,* α *is either a general clause or a positive embedded clause.*

Proposition 6. *All positive embedded programs are rigid.* [5]

Corollary 3. *Let* P *be a positive embedded program, then* M *is a answer set of* P *implies that* M *is a minimal model of* P*.*

The last part of this section presents some results about positive embedded programs that we will use to define a translation into a simpler class that does not have embedded implications. We will come back to this results in the next section, when we discuss some practical examples.

Lemma 3. *Let* P *be a positive program and let* H_P *be the set of atoms that occur in the heads of the rules of* P*. If* $P \Vdash_{\mathrm{G}_3} a$ *then* $a \in H_P$*.*

Proposition 7. *Let* P *be a logic program and let* H_P *be the set of atoms that occur in the heads of the rules of* P*. If* $a \notin H_P$*, then* $P \equiv_{\text{stable}} P \cup \{\neg a\}$

Definition 7. *Given a set of literals* L *and a positive embedded rule* r*,* r^L *is obtained from* r *by replacing all conjuncts of the form* $(a \rightarrow b)$ *by* b *if* $a \in L$*, by* $\top$ *if* $\neg a \in L$ *and doing no replacements otherwise. For any positive embedded program* P*,* P^L *is the program obtained by replacing every conjunctive rule* $r \in P$ *by* r^L*.*

Remark 2. For any positive embedded program P and a set of literals L such that $P \Vdash_{\mathrm{I}} L$, then $\vdash_{\mathrm{I}} P^L \equiv P$.

Proposition 8. *Let* P *be a positive embedded program. Let* F *be the set of facts in* P *and* $Lit(H)$ *the literals occurring in the heads of the rules in* P*. Let* $F' \subseteq F$ *and* $\neg\widetilde{H'} \subseteq \neg(\mathcal{L}_P \setminus Lit(H))$*. Then* $P \equiv_{\text{stable}} P^{F \cup \neg\widetilde{H'}}$

5 Embedded Implications in Knowledge Representation

As we have mentioned before, the need of embedded implications in the body of rules has been discovered as a relevant issue for knowledge representation in ASP. In this section, we present examples in which the definition of a problem leads us to this conclusion. If we allow implications in the body of rules, the problem can be modeled in a natural and intuitive way. The examples also show from a practical point of view how extending to the syntax can have a strong effect on the semantics of the program.

[5] Once again, applying the *redu*2 transformation to P' will only remove clauses.

Learner modeling for collaborative learning environments. In [6], ASP is used for learner modeling in collaborative learning environments. In this context, some statements similar to the following one have to be expressed in a disjunctive program: *A learner is (normally) capable of applying a knowledge element, if he is capable of applying all the knowledge elements which are a specialization of it.* We would expect a natural and intuitive translation of these statements into logic, to look like this:

$$\begin{aligned} capable(Ke) \leftarrow\ & hasSpecialization(Ke), \\ & \forall Ke1[specialization(Ke, Ke1) \rightarrow capable(Ke1)], \\ & not \sim capable(Ke). \end{aligned} \tag{1}$$

but this is certainly not a disjunctive logic programming rule. We would like to write this rule in such a way that it expresses naturally the idea we want to express and has a suitable behavior.

5.1 Semantics and Expected behavior

Now we will discuss in detail the semantics and behavior of programs modeled using embedded implications in the body of rules. Let's take, for instance, the example mentioned above. We presented one rule of the example that has implication in the body. Now we will add another rule that is also a part of the original program and give some sample facts (EDB) in order to make the example clear. Our program P would then look as follows:

$$\begin{array}{lll} knowElem(k1). & specialization(k1, k2). & capable(k2). \\ knowElem(k2). & specialization(k1, k3). & capable(k3). \\ knowElem(k3). & specialization(k1, k4). & capable(k4). \\ knowElem(k4). & specialization(k5, k6). & \\ knowElem(k5). & & \\ knowElem(k6). & & \end{array}$$

$$\begin{aligned} capable(Ke1) \leftarrow\ & specialization(Ke, Ke1), \\ & capable(Ke), \\ & not \sim capable(Ke1). \\ capable(Ke) \leftarrow\ & hasSpecialization(Ke), \\ & \forall Ke1[specialization(Ke, Ke1) \rightarrow capable(Ke1)], \\ & not \sim capable(Ke). \\ hasSpecialization(K) \leftarrow\ & knowElem(K),\ specialization(K, K1). \end{aligned} \tag{1}$$

How can we implement rule 1 in a disjunctive logic program? In some cases the need of implications in the body of rules has been addressed by translating the implication in the "classical" manner. This solution follows the tradition of Lloyd [4] and classical Logic Programming. One would be tempted to use it, since it seems natural and simple. Actually it has worked many times, but in certain cases the behavior of the new program is not intuitive, bringing us to some unexpected models. If we translate rule 1 this way, we would then replace it in P by :

$$\begin{aligned} capable(Ke) \leftarrow\ & hasSpecialization(Ke), \\ & not\ notHoldsForAll(Ke), not \sim capable(Ke). \\ notHoldsForAll(Ke) \leftarrow\ & specialization(Ke, Ke1), \\ & not\ capable(Ke1). \end{aligned}$$

The program obtained from this translation has two answer sets. One is the expected model: { *capable*($k2$), *capable*($k3$), *capable*($k4$), *capable*($k1$) }, but the other one includes *capable*($k5$), *capable*($k6$) as well, which have apparently no reason to be computed. This problem has been pointed out in other cases. For example, in the context of diagnostic reasoning in [2] a similar rule is found and the authors point out that this kind of translations does not always work.

KR with Positive Embedded Programs. Another alternative to represent our example is using class of programs we proposed in section 4.2. Here we use the universal quantifier as an abbreviation of a conjunction of elements. On the ground program, the expression $\forall x[p(x)]$ becomes a simple conjunction $p(a_1)$, $p(a_1)$ $\ldots$ $p(a_n)$ of all the ground instances of predicate p. After grounding, rule 1 is translated into a set of rules like the following ones. There is one of instance of these rules for every possible ground instance of the given predicates. In this case, there would be six rules, which correspond to all k_i such that $knowElem(k_i)$. We give only the first one as an example.

$$\begin{aligned} capable(k1) \leftarrow\ & hasSpecialization(k1), \\ & specialization(k1,k1) \rightarrow capable(k1), \\ & specialization(k1,k2) \rightarrow capable(k2), \\ & specialization(k1,k3) \rightarrow capable(k3), \\ & specialization(k1,k4) \rightarrow capable(k4), \\ & specialization(k1,k5) \rightarrow capable(k5), \\ & specialization(k1,k6) \rightarrow capable(k6), \\ & not\ capable(k1). \end{aligned}$$

We define F to be the set of facts in P and $L_H := Lit(H)$ as the set of literals that appear in the heads of the rules of P. Moreover, we can define $F_{spec} \subset F$ to be the subset of F that contains all the instances of *specialization*, and $\widetilde{L_{H\,spec}} \subset \widetilde{L_H}$ contains all instances of *specialization* that are not in L_H. Now we have the following sets:

$$\begin{aligned} F_{spec} &= \{\ specialization(k1,k2), \ldots, specialization(k5,k6)\} \\ \neg\widetilde{L_{H\,spec}} &= \{\ \neg specialization(k1,k1), \ldots \neg specialization(k6,k6)\} \end{aligned}$$

We define a new program $P' := P \cup \neg\widetilde{L_{H\,spec}}$. By Proposition 7 we know that the stable models of P are preserved in P'. Let $L := F_{spec} \cup \neg\widetilde{L_{H\,spec}}$. It is easy to see that every ground instance of *specialization* is in L, and that $P' \Vdash_{\mathrm{I}} L$. so we can apply the P'^{L} reduction to obtain a general program that has exactly the answer sets of P. We give as an example the first two rules after applying the reduction:

$$\begin{aligned} capable(k1) \leftarrow\ & hasSpecialization(k1), capable(k2), capable(k3), \\ & capable(k4), not\ capable(k1). \\ capable(k2) \leftarrow\ & hasSpecialization(k2), not\ capable(k2). \end{aligned}$$

The only answer set of the general program we obtain corresponds to the one we expected: { *capable*($k2$), *capable*($k3$), *capable*($k4$), *capable*($k1$) }. Remember that since positive embedded programs are rigid, we can assure that even when the extensional database is changed, the answer sets are always going to be minimal models as well. From a practical perspective, this simple translation allows us to model problems with implication in the body of rules and implement them with the standard software.

6 Conclusions and Related Work

Extending the ASP semantics to wider classes of formulas allows us to represent problems in a more natural way. The knowledge representation community is always on the search of ways to represent problems and explicit knowledge with increasing naturality. As we have mentioned, some real life statements lose their natural behavior if no implications in the body are allowed. We believe that extending the semantics of ASP to programs with embedded implications in the body of the rules can be an effective contribution in this direction. Our motivation and examples are in real life applications, in areas where we could actually use logic programming as a reasoning component of intelligent systems.

To the best of our knowledge, an answer sets semantics for programs with embedded implications had not been defined. The characterization we have used [9] is valid for arbitrary theories, but it is presented in the context of augmented programs. We have noticed, however, that some of the examples that we have modeled with this extended syntax could also be represented using parametric connectives due to Perri and Leone [12]. We can point out that the semantics here defined do not have the stratification restriction they present. The relations between both approaches is subject of deeper study, but it seems that in some cases both semantics coincide.

References

1. Chitta Baral. *Knowledge Representation, reasoning and declarative problem solving with Answer Sets.* Cambridge University Press, Cambridge, 2003.
2. Michael Gelfond and Marcelo Balduccini. Diagnostic reasoning with a-prolog. *TPLP*, 3(4-5):425–461, 2003.
3. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
4. John W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, second edition, 1987.
5. Juan Antonio Navarro. Answer set programming through G_3 logic. In Malvina Nissim, editor, *Seventh ESSLLI Student Session, European Summer School in Logic, Language and Information*, Trento, Italy, August 2002.
6. Magdalena Ortiz de la Fuente. An application of answer sets programming for supporting collaboration in agent-based cscl enviroments. In Balder Ten Cate, editor, *ESSLLI03 Student Session. European summer School of Logic, Language and Information*, Vienna, Austria, August 2003.
7. Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Equivalence in answer set programming. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation. 11th International Workshop, LOPSTR 2001*, number 2372 in LNCS, pages 57–75, Paphos, Cyprus, November 2001. Springer.
8. Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. A logical approach for A-Prolog. In Ruy de Queiroz, Luiz Carlos Pereira, and Edward Hermann Haeusler, editors, *9th Workshop on Logic, Language, Information and Computation (WoLLIC)*, volume 67 of *Electronic Notes in Theoretical Computer Science*, pages 265–275, Rio de Janeiro, Brazil, 2002. Elsevier Science Publishers.
9. Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Applications of intuitionistic logic in answer set programming. Accepted to appear at the TPLP journal, 2003.
10. David Pearce. From here to there: Stable negation in logic programming. In D. M. Gabbay and H. Wansing, editors, *What Is Negation?*, pages 161–181. Kluwer Academic Publishers, Netherlands, 1999.
11. David Pearce. Stable inference as intuitionistic validity. *Logic Programming*, 38:79–91, 1999.
12. Simona Perri and Nicola Leone. Parametric connectives in disjunctive logic programming. In *ASP03 Answer Set Programming: Advances in Theory and Implementation*, Messina, Sicily, September 2003.