View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCL Discovery

Computing Preferred Safe Beliefs

Luis Angel Montiel Juan Antonio Navarro

Departamento de Sistemas Computacionales, Universidad de las Américas - Puebla
{is110493,ma108907}@mail.udlap.mx

Abstract

We recently proposed a definition of a language for nonmonotonic reasoning based on intuitionistic logic. Our main idea is a generalization of the notion of answer sets for arbitrary propositional theories. We call this extended framework *safe beliefs*. We present an algorithm, based on the Davis-Putnam (DP) method, to compute safe beliefs for arbitrary propositional theories. We briefly discuss some ideas on how to extend this paradigm to incorporate preferences.

Keywords: Answer Sets, Davis Putnam, Safe Beliefs, Preferences, Algorithms.

1 Introduction

A-Prolog (Stable Logic Programming [9] or Answer Set Programming [11]) is the realization of much theoretical work on Nonmonotonic Reasoning and AI applications of Logic Programming (LP) in the last 15 years. This is an important logic programming paradigm that has now great acceptance in the community. Efficient software to compute answer sets and a large list of applications to model real life problems justify this assertion. The two most well known systems that compute answer sets are DLV[1] and SMODELS[2].

We recently proposed a generalization of the notion of answer sets for arbitrary propositional theories [18]. We call this extended framework *safe beliefs*. A first advantage of this proposed approach is the fact that the new definition does not imply any particular restriction in the form or syntax that logic programs should have. This definition would allow, for instance, to use embedded implications inside clauses contained in logic programs. We observe that this broader syntax could allow us to write some clauses in logic programs using a more natural form.

The Davis-Putnam (DP) [5] method is one of the major practical methods for the satisfiability (SAT) problem of classical propositional logic. In the last

[1] http://www.dbai.tuwien.ac.at/proj/dlv/
[2] http://www.tcs.hut.fi/Software/smodels/

three years, Hantao Zhang (and others) developed a very efficient implementation of the Davis-Putnam method called SATO [22]. In this paper we propose a method to compute safe beliefs for arbitrary theories based on the Davis Putnam method. Recent research [2], has shown that when the stable semantics corresponds to the *supported semantics*, a satisfiability solver like SATO can be used to compute stable models. Interestingly, some examples are presented in [2] where the running time of SATO is approximately ten times faster than SMODELS.

For this reason, we believe that our proposed approach is justified. It has been reported in [4], from the creators of the DLV system, an approach also based in the DP method. Their algorithm works, however, proposing candidate models. Those candidate models are then checked to see if they are actually answer sets and then reported.

In our approach we construct answer sets directly. The key point is that, the splitting part of our algorithm adds $\neg\neg a$ instead of just a, which is formally justified by our results [18]. We remark that our approach can be used to obtain the set of safe beliefs of any propositional theory.

We also extend our notion of safe beliefs to incorporate preferences. We call our augmented paradigm preference safe beliefs, and we also show how to compute models under this paradigm.

In this paper we restrict our attention to finite propositional theories, the semantics can be extended to theories with variables by grounding[3]. This is a standard procedure in A-Prolog. We assume that the reader has some basic background in logic and A-Prolog.

2 Background

In this section we review some basic concepts and definitions that will be used along this paper. We introduce first the language of propositional logic, and define some classes of programs typically used in the context of logic programming. Finally we make some comments on intuitionistic logic that is used to provide a logical framework for logic programming and nonmonotonic reasoning.

2.1 Propositional Logic

We use the set of propositional formulas in order to describe rules and information within logic programs. Formally we consider a language built from an alphabet containing: a denumerable set $\mathcal{L}$ of elements called *atoms* or *atomic formulas*; the binary connectives $\wedge$, $\vee$ and $\rightarrow$ to denote conjunction, disjunction and implication respectively; the unary connective $\neg$ for negation; the 0-ary connectives $\bot$ and $\top$ to denote falsity and truth; and auxiliary symbols (,).

Formulas can be constructed as usual in logic. In fact, the formula $\neg F$ can be alternatively introduced as an abbreviation of $F \rightarrow \bot$, and $\top$ as $\bot \rightarrow \bot$. We

[3]Without function symbols to ensure that a ground program would be finite.

can write, as usual, $F \leftrightarrow G$ to denote the formula $(F \rightarrow G) \wedge (G \rightarrow F)$. Also, the formula $G \leftarrow F$ is just another way of writing $F \rightarrow G$.

A *theory* is a set of formulas, we restrict our attention to finite theories. For a given theory T its *signature*, denoted $\mathcal{L}_T$, is the set of atoms that occur in the theory T. Observe that, since we consider finite theories, their signatures are also finite. Given a theory T we also define the negated theory $\neg T = \{\neg F \mid F \in T\}$. A *positive theory* is one that does not contain occurrences of the connectives $\bot$ and $\neg$.

We say that the *weak occurrences* of an atom inside a theory are those occurrences of the atom that lie inside the scope of a negation connective, i. e. all the occurrences of atoms in a subformula $\neg F$ are weak.

2.2 Logic Programs

In our context a *logic program* is just a propositional theory. We can think, in fact, of the words *theory* and *program* as synonyms; we use the former when stating general results in mathematical logic, and the latter when dealing with logic programs having a particular syntax and/or semantics.

The syntax of formulas within logic programs is usually defined in terms of special formulas know as clauses. A *clause* is, in general, a formula $H \leftarrow B$ with implication as the principal connective. The formulas H and B are known as the *head* and *body* of the clause respectively. The special case of a clause with the form $\bot \leftarrow B$ is known as a *constraint*. Each formula H, whose principal connective is not implication, is know as a *fact* and is associated with the clause $H \leftarrow \top$.

We introduce now some of the classes of programs commonly found in the literature. An *augmented clause* is a clause where both H and B can be arbitrary logic formulas built from the connectives $\wedge$, $\vee$ and $\neg$ arbitrarily nested. Note, however, that embedded implications are not allowed in augmented clauses. An *augmented program* is a logic program that contains only augmented clauses.

A *disjunctive clause* is built from a (non empty) disjunction of atoms in the head and a conjunction of literals in the body. A disjunctive clause has then the form

$$h_1 \vee \cdots \vee h_n \leftarrow b_1 \wedge \cdots \wedge b_m \wedge \neg b_{m+1} \wedge \cdots \wedge \neg b_{m+l}.$$

where each h_i and b_j is an atom, $n \geq 1$, $m \geq 0$ and $l \geq 0$. A *disjunctive program* is then defined as a program containing only disjunctive clauses. Observe that, in particular, constraints are not allowed inside disjunctive programs.

Example 2.1. The following are examples of the clauses just defined

$a \leftarrow (b \rightarrow c)$	propositional
$\neg(p \wedge \neg q) \leftarrow a \vee (\neg b \wedge c).$	augmented
$a \vee b \leftarrow c \wedge d \wedge \neg e.$	disjunctive

2.3 Notation on Intuitionistic Logic

Intuitionistic logic provides the logic foundations of safe beliefs. This logic can be defined in terms of Hilbert type proof systems of axioms and inference rules. Alternative definitions can be given in terms of Kripke models.

We use the standard notation $\vdash F$ to denote that F is a provable formula, an intuitionistic theorem. If T is a theory we understand the symbol $T \vdash F$ to mean that $\vdash (F_1 \wedge \cdots \wedge F_n) \rightarrow F$ for some F_i contained in T. This is not the usual definition given in literature, but can be shown to be equivalent because of results like *Deduction Theorem*. Similarly, if U is a theory, we use the symbol $T \vdash U$ to denote $T \vdash F$ for every $F \in U$. We use $Cn(T)$ to denote the set of all logical consequences of T, i.e. the set $\{F \mid T \vdash F\}$.

A theory T is said to be consistent, with respect to intuitionistic logic, if it is not the case that $T \vdash \bot$. We say that a theory T is (literal) complete iff, for all $a \in \mathcal{L}_T$, we have either $T \vdash a$ or $T \vdash \neg a$. We use the notation $T \Vdash U$ to stand for the phrase: T is consistent and $T \vdash U$. Finally we say that two theories T_1 and T_2 are equivalent, under intuitionistic logic, denoted by $T_1 \equiv T_2$, if it is the case that $T_1 \vdash T_2$ and $T_2 \vdash T_1$.

3 Logical Foundations for A-Prolog

A popular semantic operator for logic programs is the answer set semantics. Some of the main features of answer sets are its nonmonotonicity and the integration of negation as failure[4]. This sort of features are extremely useful to deal with concepts such as default knowledge and modeling inertial rules. The answer set semantics constitutes the formal basement of the A-Prolog programming paradigm and proved to be particularly useful for several real life and research applications.

In the original definition of answer sets [9, 11], however, the relation of the semantics with a particular logic is not entirely clear and seems to be hidden under an *ad hoc* setting of reductions and minimal models. Safe Beliefs were introduced in [20] in order to provide logical foundations to the A-Prolog paradigm.

Consider the case of a logic agent. The task of such an agent is, when provided with some initial knowledge describing its world or application domain, to do inference and produce new knowledge. We would like our agent, for instance, to be able to answer queries and solve problems in its application domain.

We can use a propositional theory T to represent the initial knowledge of our agent. Under the premise that T is consistent, it makes sense to say that the agent *knows* F if the formula F is an intuitionistic consequence of the theory T.

[4]The negation symbol we use ($\neg$) will play the role of the negation as failure in logic programs. The authors in [11] use, however, the symbol *not* instead.

The use of intuitionistic logic, a logic of knowledge, as the underlying inference system seems a natural choice for this approach.

But we also want our agent to do nonmonotonic inference. Informally speaking we will allow our agent to *guess* or *suppose* things in order to make more inference. We must be cautious, however, since we don't wan't our agent to precipitate conclusions or make unnecessary assumptions. The agent should only *suppose* facts if they are helpful to provide new knowledge. The following definition formalizes this idea.

Definition 3.1. Let T be a theory and let E be a theory with $\mathcal{L}_E \subseteq \mathcal{L}_T$. We say that the theory E is an *explanation* of T if the theory $T \cup \neg\neg E$ is both consistent and complete (w.r.t. intuitionistic logic).

A very natural reading of this definition, in the context of our logic agent, would be that: *"A theory E is an explanation of the agent's world if (i) it is consistent with its initial knowledge and (ii) supposing that E is true is enough to answer any possible question in its domain."* This *supposing* corresponds to the double negation in the intuitionistic proof system used as the underlying inference system.

The agent is allowed now to *believe* things in order to obtain a complete explanation of its world. If E is an explanation of T, it makes sense to say that the agent *believes* F (assuming E) if the formula F is an intuitionistic consequence of $T \cup \neg\neg E$. There can also be, of course, several explanations for a given initial knowledge. Brave and cautions beliefs can be defined in the natural way. The agent *bravely believes* a formula if there is an explanation that makes the formula true, and *cautiously believes* a formula if that formula is true under all possible explanations of its initial knowledge.

The explanations for a given theory naturally define semantics for logic programs, we call this the *safe belief semantics*.

Definition 3.2. Let T be a theory. If E is an explanation of T then the set of atoms $Cn(T \cup \neg\neg E) \cap \mathcal{L}_T$ is a *safe belief* of T.

Note that, by definition, safe beliefs are subsets of the atoms that occur in the theory T. Moreover, if M is a safe belief of T, it can be easily shown that $E = M \cup \neg(\mathcal{L}_T \setminus M)$ is an explanation of T with $Cn(T \cup \neg\neg E) \cap \mathcal{L}_T = M$.

Despite the apparent lack of a logical intuition in the definition of answer sets, it has been proved that they actually coincide with safe beliefs (in the class of augmented programs). This is not only a significant property of safe beliefs, but also serves to provide solid logical foundations for the A-Prolog programming paradigm and allows natural generalizations.

Theorem 3.1. *[20] Let P be an augmented program. A set of atoms M is a safe belief of P if and only if M is an answer set of P.*

Observe that our definition of safe beliefs does not impose any particular restriction in the form or syntax that the logic programs should have. In our particular case this definition allows the use of embedded implications inside

formulas, a feature that is not very common in current logic programming paradigms. Augmented programs, for instance, do not allow embedded implications. The use of the full set of propositional formulas can be useful to describe problems in a more convenient and natural way.

We even suspect that this unrestricted syntax can be helpful to model concepts like aggregation in logic programs, as the ones described in [14, 15]. Some other potential applications, suggested by M. Gelfond (e-mail communication), could be found in the field of action languages.

Another important property of safe beliefs is that its definition naturally follows from provability relations in a well known mathematical logic, namely intuitionistic logic, providing solid foundations for our approach. The relation with logic is more clear and direct than, for instance, in the original definition of answer sets. In [20] it has also been proved, in fact, that any proper intermediate logic can be used, instead of intuitionistic logic, to construct safe beliefs defining exactly the same semantics.

Equivalence notions can be easily described in terms of logic. It has been shown that logic G_3 characterizes[5] strong equivalence for safe beliefs [12]. We have also shown for instance in [1, 17] how the logic G_3 can be used to debug a safe belief program by taking advantage of the 3-valued nature of G_3. Although the notion of safe beliefs is quite recent, some interesting results and applications have already been found.

4 Reduction of Theories

In this section we will define some reductions that can be applied to theories in order to simplify their structure. The notion of reductions and/or transformations has several applications in Logic Programming; see for instance [6, 16, 21]. Some properties of these reductions will be studied.

Definition 4.1. For a formula F, we define its reduction with respect to the $\bot$ and $\top$ symbols, denoted $\text{Reduce}_\bot(F)$, by replacing each subformula in F according to the following rules:

1. Conjunction:

$$A \wedge \bot \text{ or } \bot \wedge A \text{ with } \bot.$$
$$A \wedge \top \text{ or } \top \wedge A \text{ with } A.$$

2. Disjunction:

$$A \vee \bot \text{ or } \bot \vee A \text{ with } A.$$
$$A \vee \top \text{ or } \top \vee A \text{ with } \top.$$

3. Implication:

[5] Logic G_3 is the 3-valued logic defined by Gödel, also known as the logic of Here and There (HT).

$A \rightarrow \top$ or $\bot \rightarrow A$ with $\top$.
$\top \rightarrow A$ with A.
$A \rightarrow \bot$ with $\neg A$.

4. Negation:

$$\neg\bot \text{ with } \top; \text{ and } \neg\top \text{ with } \bot.$$

This rules should be applied until no subformula in F matches any of the patterns presented above.

For a given theory T we can extend this definition as follows: if there is a formula $F \in T$ with $\text{Reduce}_\bot(F) = \bot$ then $\text{Reduce}_\bot(T) = \{\bot\}$; otherwise $\text{Reduce}_\bot(T)$ is defined as the theory:

$$\{\text{Reduce}_\bot(F) \mid F \in T, \text{Reduce}_\bot(F) \neq \top\} .$$

Note that if every formula of the theory reduces to $\top$ then the entire theory reduces to the empty theory. Also note that this reduction satisfies equivalence with respect to intuitionistic logic, that is $T \equiv \text{Reduce}_\bot(T)$. Consequently the equivalence is also true for classical and any intermediate logic.

Definition 4.2 (Negative Red.). [18] Given a theory T and a set of atoms M replace in T all occurrences of those atoms in M with the symbol $\bot$ to obtain a new theory T'. The *negative reduction* of T with respect to M, denoted by $\text{Reduce}_\mathcal{N}(T, M)$, is defined as $\text{Reduce}_\bot(T')$.

Definition 4.3 (Weak Positive Red.). [18] Given a theory T and a set of atoms M replace in T all weak occurrences of those atoms in M with the symbol $\top$ to obtain a new theory T'. The *weak positive reduction* of T with respect to M, denoted by $\text{Reduce}_\mathcal{W}(T, M)$, is defined as $\text{Reduce}_\bot(T')$.

Note that, while computing the weak positive reduction of a theory, new negation connectives can be introduced (i.e. while reducing an implication) modifying the "weak" status of some atoms in the theory. The reduction then should be applied several times until no weak occurrences of atoms in M appear in the theory T.

Definition 4.4. Given a theory T and two disjoint sets of atoms $M_\mathcal{W}$, $M_\mathcal{N}$ we can define the *complete reduction* of T, denoted by $\text{Reduce}(T, M_\mathcal{W}, M_\mathcal{N})$, as the theory $\text{Reduce}_\mathcal{W}(\text{Reduce}_\mathcal{N}(T, M_\mathcal{N}), M_\mathcal{W})$.

Example 4.1. The following example illustrates our definitions of negative and weak positive reduction. Suppose that $M_\mathcal{N} = \{b\}$ and $M_\mathcal{W} = \{a, c\}$, and consider the theory T:

$a \vee c \leftarrow \neg b \wedge \neg c.$
$\bot \leftarrow b \wedge c.$
$a \leftarrow (\neg b \rightarrow c).$

then $T_1 = \text{Reduce}_{\bot}(T, M_{\mathcal{N}})$ will become:

$$\begin{array}{lcl} a \vee c \leftarrow \neg\bot \wedge \neg c. & \mapsto & a \vee c \leftarrow \neg c. \\ \bot \leftarrow \bot \wedge c. & \mapsto & \top. \\ a \leftarrow (\neg\bot \rightarrow c). & \mapsto & a \leftarrow c. \end{array}$$

and $T_2 = \text{Reduce}_{\mathcal{W}}(T_1, M_{\mathcal{W}})$:

$$\begin{array}{lcl} a \vee c \leftarrow \neg\top. & \mapsto & \top. \\ a \leftarrow c. & \mapsto & a \leftarrow c. \end{array}$$

to finally obtain $T_2 = \{a \leftarrow c\}$.

This complete reduction also satisfy an equivalence condition with respect to intuitionistic logic. Namely, given a theory T and two sets of atoms $M_{\mathcal{W}}$ and $M_{\mathcal{N}}$, the theory $\text{Reduce}(T, M_{\mathcal{W}}, M_{\mathcal{N}}) \cup \neg\neg M_{\mathcal{W}} \cup \neg M_{\mathcal{N}}$ is equivalent to $T \cup \neg\neg M_{\mathcal{W}} \cup \neg M_{\mathcal{N}}$.

Another important property of this reduction is that, if the condition $\mathcal{L}_T = M_{\mathcal{W}} \cup M_{\mathcal{N}}$ holds, then the theory $\text{Reduce}(T, M_{\mathcal{W}}, M_{\mathcal{N}})$ is, if not reduced to $\{\bot\}$, always a positive theory.

Other well known reductions that appear in literature can be applied to further simplify a theory, see for instance [8, 19]. Due, to lack of space, we omit a discussion of them in this paper.

5 Computing Safe beliefs

The Davis-Putnam (DP) method is one of the major practical methods for the satisfiability (SAT) problem of classical propositional logic. Hantao Zhang (and others) developed a very efficient implementation of the Davis-Putnam method called SATO [22]. Recent research has shown how this kind of solvers can be used, under certain conditions, to efficiently compute stable models for logic programs. Examples had been presented where the running time of SATO is ten time faster than the leading stable model finder SMODLES.

An alternative, suggested in [13], considers the reduction of an arbitrary theory (using a polynomial translation also presented in [13]) to a more simple disjunctive program and then use an answer set finder (such as DLV or SMODELS) to finally compute safe beliefs.

The problem with this sort of translations, and others following a similar approach, is that they may introduce many new atoms in order to perform reductions "*thus greatly widening the space of assignments in which the DP procedure has to search in order to find the solutions*", as E. Giunchiglia and R. Sebastiani [10] have already pointed out.

In this section we present a method to compute safe beliefs for arbitrary theories based on the Davis-Putnam method. We will avoid the introduction of new atoms by using the set of reductions presented in Section 4 that, instead of trying to reduce the structure of the program, are helpful to evaluate the effect

of the decisions made by the DP method reducing the size of the evaluated program.

Our algorithm, sketched in the function GetSafeBeliefs takes as input a program and two sets of atoms $M_{\mathcal{W}}$ and $M_{\mathcal{N}}$ that contains atoms that can be supposed as positive or negative respectively. The algorithm first reduces the program using the information contained in $M_{\mathcal{W}}$ and $M_{\mathcal{N}}$, and then it checks for some easy conditions that can early reject a model thus avoiding unnecessary branching. Otherwise, the function is called recursively guessing the value of a new atom returning a set containing all the safe beliefs found. As in [10], we apply the splitting part of the algorithm only for the atoms that occur in the original program.

function GetSafeBeliefs$(P, M_{\mathcal{W}}, M_{\mathcal{N}})$
  $P \longleftarrow$ Reduce$(P, M_{\mathcal{W}}, M_{\mathcal{N}})$.
  if $P = \{\bot\}$ **or** $\mathcal{L}_P \subset M_{\mathcal{W}}$ **then**
    return $\emptyset$.
  else if $\mathcal{L}_P = M_{\mathcal{W}}$ **then**
    return CheckPositive(P).
  else
    Let x be an atom from $\mathcal{L}_P \setminus M_{\mathcal{W}}$.
    return
      GetSafeBeliefs$(P, M_{\mathcal{W}} \cup \{x\}, M_{\mathcal{N}})$
      $\cup$ GetSafeBeliefs$(P, M_{\mathcal{W}}, M_{\mathcal{N}} \cup \{x\})$.
  end if
end function

The function GetSafeBeliefs is trying to guess, using the DP method, for each atom whether it can be assumed to be positive or negative. When no more guesses are possible, the function CheckPositive is used to finally determine if the set $M_{\mathcal{W}}$ is an safe belief of the reduced theory T or not.

Then the function CheckPositive takes as input a theory T. Recall that, by the properties of the reductions applied, T is always a positive theory. The function must return $\{\mathcal{L}_T\}$ is $T \vdash \mathcal{L}_T$ and $\{\}$ otherwise.

The problem of determining, for a given positive theory T, if $T \vdash \mathcal{L}_T$ seems to be an NP problem. It is easy to find reductions of the *Check Positive* problem to other NP-complete problems such as finding minimal models, or even the SAT problem. A similar DP method can also be used to solve this problem trying to find a model $M \subset \mathcal{L}_T$ (proper inclusion) for the theory T. If such model exists then it is false that $T \vdash \mathcal{L}_T$, otherwise it constitutes a proof of the fact that $T \vdash \mathcal{L}_T$.

At the point of the algorithm when it requires to select an atom from $\mathcal{L}_P \setminus M_{\mathcal{W}}$, heuristics can be used to improve its performance. For instance, it is possible to select the atom with more occurrences in the program P.

Theorem 5.1 (Correctness). *The function* GetSafeBeliefs$(T, \emptyset, \emptyset)$ *terminates and returns the set of safe beliefs of the theory* T.

Proof. (Sketch) It is immediate that the algorithm terminates since the signature of P is finite. The function CheckPositive is assumed to be correct and return the corresponding set of safe beliefs. The final case solves the problem recursively adding new atoms to the partially computed safe beliefs in $M_{\mathcal{W}}$ and $M_{\mathcal{N}}$. □

6 Preferred Safe Beliefs

A useful topic of interest in answer set programming is to allow preferences, see [3, 7]. Thus, we extend our safe beliefs paradigm to express a simple but still useful form of preferences. Moreover, our algorithm presented in previous section can be easily adjusted to compute such preferable models.

Consider the following basic example. Suppose that there are courses a, b and c available. If one course is closed, the system adds the constraint $\neg x$ (where x is the course). Now consider a student asking to take course a and either b or c. The student however prefers to take course b over c. The preference program would be the next one:

$$b > c$$
$$a \wedge (b \vee c)$$

Here the expression $b > c$ denotes the preference of b over c. The preferable safe belief of this program is $\{a, b\}$, assuming all courses were open. If b gets closed, the program becomes:

$$b > c$$
$$a \wedge (b \vee c)$$
$$\neg b$$

This new program has, as expected, the preferable safe belief $\{a, c\}$.

The main idea, of our proposed preference semantics, is to extend the syntax of logic programs to allow an expression of the form: $a_1 > \cdots > a_n$, where $a_1, \ldots, a_n$ are different atoms in the language.

This preference rule naturally induces an order relation among the safe beliefs of the given logic program. The preferred safe beliefs is then the first model obtained with respect to this induced order.

Also observe that this extension to the semantic of safe beliefs allows to specify the preference for a given formula F, perhaps describing some desired condition to hold in the expected models. Simply add a new atom f at the desired position in the expression $a_1 > \cdots > a_n$ and, using the extended syntax of safe beliefs, incorporate the rule $f \leftrightarrow F$ to the original program.

The algorithm presented previously can be easily adapted to this preference semantics. The defined order can be used to choose the most preferred atom, still occurring in the program, to perform the next split in the DP algorithm; and halt the recursion as soon as a model is found.

7 Conclusions

We presented a variant of the Davis Putnam method to compute safe beliefs. To the best of our knowledge our algorithm is original. Furthermore, we do not know of any algorithm to compute safe beliefs (or any similar extension of answer sets) for arbitrary propositional theories.

Further research needs to be done in order to incorporate, for instance, several ideas to improve efficiency as those discussed in [4]. We extended the notion of safe beliefs to model a simple but useful form of preferences among safe beliefs. Finally we show how to compute models of our extended language.

References

[1] Juan Carlos Acosta Guadarrama, José Arrazola, and Mauricio Osorio. Making belief revision with LUPS. In Juan Humberto Sossa Azuela and Gustavo Arroyo Figueroa, editors, *XI International Conference on Computing*, México, D.F., November 2002. CIC-IPN.

[2] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming. In Chitta Baral and Mirek Truszczynski, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, Breckenridge, Colorado, USA, April 2000.

[3] Gerhard Brewka. Logic programming with ordered disjunction. In Ulrich Junker, editor, *Preferences in AI and CP: Symbolic Approaches, Papers from the AAAI Workshop*, pages 1–8, Edmonton, Alberta, Canada, August 2002. Extended version presented at NMR-02.

[4] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pruning operators for answer set programming systems. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning*, pages 200–209, 2002.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *ACM*, 7:201–215, 1960.

[6] Jürgen Dix, Mauricio Osorio, and Claudia Zepeda. A general theory of confluent rewriting systems for logic programming and its applications. *Annals of Pure and Applied Logic*, 108(1–3):153–188, 2001.

[7] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Computing preferred and weakly preferred answer sets by meta-interpretation in answer set programming. In Alessandro Provetti and Tran Cao Son, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 45–52, Stanford, USA, 2001. AAAI Press.

[8] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplyfying logic programs under uniform and strong equivalence. In Ilkka Niemelä and Vladimir Lifschitz, editors, *7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, number 2923 in LNCS, pages 87–99, Florida, USA, January 2004. Springer.

[9] 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.

[10] Enrico Giunchiglia and Roberto Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In Evelina Lamma and Paola Mello, editors, *Advances in Artificial Intelligence, 6th Congress of the Italian Association for Artificial Intelligence*, number 1792 in LNAI, pages 84–94. Springer, 1999.

[11] Vladimir Lifschitz, L. R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[12] 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.

[13] Juan Antonio Navarro. Properties of translations for logic programs. In Balder ten Cate, editor, *Eight ESSLLI Student Session, European Summer School in Logic, Language and Information*, Vienna, Austria, August 2003.

[14] Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New generation computing*, 17(3):255–284, 1999.

[15] Mauricio Osorio, Bharat Jayaraman, and David Plaisted. Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238, 1999.

[16] 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.

[17] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Debugging in A-Prolog: A logical approach (abstract). In P.J. Stuckey, editor, *Logic Programming. 18th International Conference, ICLP 2002*, number 2401 in LNCS, pages 482–483, Copenhagen, Denmark, August 2002. Springer.

[18] 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.

[19] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Applications of intuitionistic logic in answer set programming. Accepted to appear at the TPLP journal, 2003.

[20] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Safe beliefs for propositional theories. Accepted to appear at ELSEVIER, 2003.

[21] Mauricio Osorio, Juan Carlos Nieves, and Chis Giannella. Useful transformations in answer set programming. In Alessandro Provetti and Tran Cao Son, editors, *Proceedings of the American Association for Artificial Intelligence (AAAI) 2001 Spring Symposium Series*, pages 146–152, Stanford, E.U., 2001. AAAI Press.

[22] Hanato Zhang. SATO: A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, 22:1–3, March 1993.