On the Existence of Answer Sets in Normal Extended Logic Programs

Martin Caminada[1] and **Chiaki Sakama**[2]

Abstract. One of the serious problems in *answer set programming* is that relatively small pieces of information can cause a total absence of answer sets. To cope with this problem, this paper introduces a class of *normal extended logic programs* which are extended logic programs, whose defeasible rules are comparable to normal defaults in default logic. Under suitable program transformations, we show that every normal extended logic program always yields at least one answer set.

1 Introduction

Answer set programming (ASP) is a declarative programming paradigm which is useful for AI problem solving [1, 4]. In ASP a program is represented as an *extended logic program* whose declarative meaning is given by the *answer set semantics* [3]. However, it is well-known that a program does not always have an answer set. A notorious example is the rule like $\mathtt{p} \leftarrow \mathtt{not}\,\mathtt{p}$. The existence of such a "negative loop" – an atom depending on its default negation, can cause a total absence of answer sets, and blocks all useful inferences from other parts of the program.

From the knowledge representation viewpoint, a negative loop is often considered anomalous information. So there would be a good reason that a program including such anomalous information is unusual anyway. The problem, however, is that even programs that do not include any such anomalous information often fail to have an answer set. This is illustrated by, for instance, the "Married John" example from [2]: (a) "John wears something that looks like a wedding ring." (b) "John parties with his friends until late." (c) "Someone wearing a wedding ring is usually married." (d) "A party-animal is usually a bachelor." (e) "A married person, by its definition, has a spouse." (f) "A bachelor, by definition, does not have a spouse." These sentences are represented by the following program:

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{r} & \leftarrow & & \mathtt{p} & \leftarrow & \\
\mathtt{m} & \leftarrow & \mathtt{r}, \mathtt{not}\,\neg\,\mathtt{m} & \mathtt{b} & \leftarrow & \mathtt{p}, \mathtt{not}\,\neg\,\mathtt{b} \\
\mathtt{hs} & \leftarrow & \mathtt{m} & \neg\mathtt{hs} & \leftarrow & \mathtt{b}\,.
\end{array}$$

This program contains no negative cycle, and encodes given information in a natural way. The program has no answer set, however.

One may consider that interpreting strict (NAF-free) rules as *clauses* in propositional logic and representing them as disjunctive facts would solve the problem. Rewriting the strict rules as disjunctive facts in the above program would yield the following.

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{r} & \leftarrow & & \mathtt{p} & \leftarrow & \\
\mathtt{m} & \leftarrow & \mathtt{r}, \mathtt{not}\,\neg\,\mathtt{m} & \mathtt{b} & \leftarrow & \mathtt{p}, \mathtt{not}\,\neg\,\mathtt{b} \\
\mathtt{hs} \vee \neg\mathtt{m} & \leftarrow & & \neg\mathtt{hs} \vee \neg\mathtt{b} & \leftarrow & .
\end{array}$$

This program has two answer sets: $\{\mathtt{r}, \mathtt{p}, \mathtt{m}, \neg\mathtt{b}, \mathtt{hs}\}$ and $\{\mathtt{r}, \mathtt{p}, \mathtt{b}, \neg\mathtt{m}, \neg\mathtt{hs}\}$. Unfortunately, such rewriting does not work in general. For instance, the following program contains only normal defeasible rules and disjunctive facts but still has no answer set.[3]

$$\begin{array}{rcl@{\qquad}rcl}
\neg\mathtt{a} & \leftarrow & \mathtt{b}, \mathtt{not}\,\mathtt{a}, & \mathtt{b} & \leftarrow & \mathtt{a}, \mathtt{not}\,\neg\,\mathtt{b}, \\
\mathtt{c} & \leftarrow & \neg\,\mathtt{a}, \mathtt{not}\,\neg\,\mathtt{c}, & \mathtt{d} & \leftarrow & \mathtt{c}, \mathtt{not}\,\neg\,\mathtt{d}, \\
\mathtt{a} & \leftarrow & \mathtt{d}, \mathtt{not}\,\neg\,\mathtt{a}, & \mathtt{a} \vee \mathtt{b} \vee \mathtt{c} & \leftarrow & .
\end{array}$$

The above discussion indicates that specifying syntactic restrictions under which an extended logic program is guarenteed to have answer sets is not a trivial task, especially when one also want to preserve a form of strict reasoning.

2 Basic Definitions

A *program* considered in this paper is an *extended logic program* (ELP) [3], which is a finite set of *rules* of the form:

$$\mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{not}\,\mathtt{b}_1, \ldots, \mathtt{not}\,\mathtt{b}_m \qquad (1)$$

where each $\mathtt{c}$, $\mathtt{a}_i$ and $\mathtt{b}_j$ is a positive/negative literal and $\mathtt{not}$ stands for *default negation* or *negation as failure* (NAF). $\mathtt{not}\,\mathtt{b}_j$ is called an *NAF-literal*. If $\mathtt{a}$ is an atom, we identify $\neg\neg\mathtt{a}$ with $\mathtt{a}$. The literal $\mathtt{c}$ is the *head* and the conjunction $\mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{not}\,\mathtt{b}_1, \ldots, \mathtt{not}\,\mathtt{b}_m$ is the *body*. The head is nonempty, while the body is possibly empty. For each rule r of the form (1), $head(r)$ represents the literal c, and $body^+(r)$ and $body^-(r)$ represent the sets $\{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ and $\{\mathtt{b}_1, \ldots, \mathtt{b}_m\}$, respectively. A rule is called *strict* if it is of the form:

$$\mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n\,. \qquad (2)$$

Otherwise, a rule (1) is called *defeasible*. Given a program P, we use the notation $strict(P)$ for the set of all strict rules of P, and $defeasible(P)$ for the set of all defeasible rules of P. Clearly, $P = strict(P) \cup defeasible(P)$. A program is *NAF-free* if it consists of strict rules only. A program (rule, literal) is *ground* if it contains no variable. Throughout the paper, we handle finite ground programs unless stated otherwise.

The semantics of ELPs is given by the *answer set semantics* [3]. Let Lit be the set of all ground literals in the language of a program. A set $S (\subseteq Lit)$ *satisfies* a ground rule r if $body^+(r) \subseteq S$ and $body^-(r) \cap S = \emptyset$ imply $head(r) \in S$. S satisfies a program P if

[1] Institute of Information and Computing Sciences, P.O. Box 80 089 3508 TB Utrecht The Netherlands; email: martinc@cs.uu.nl

[2] Department of Computer and Communication Sciences, Wakayama University, Wakayama 640-8510, Japan; email: sakama@sys.wakayama-u.ac.jp

[3] We invite those who claim to have found one to verify the minimality of their solution.

S satisfies every rule in P. Let P be an NAF-free ELP. Then, a set $S(\subseteq Lit)$ is an *answer set* of P if S is a minimal set such that (i) S satisfies every rule from P; and (ii) if S contains a pair of complementary literals L and $\neg L$, $S = Lit$. Next, let P be any ELP and $S \subseteq Lit$. For every rule r of P, the rule $head(r) \leftarrow body^{+}(r)$ is included in the *reduct* P^S if $body^{-}(r) \cap S = \emptyset$. Then, S is an *answer set* of P if S is an answer set of P^S. An answer set is *consistent* if it is not Lit. A program P is *consistent* if it has a consistent answer set; otherwise P is *inconsistent*. Remark that, by the definition, an ELP P has the answer set Lit iff $strict(P)$ has the answer set Lit.

3 Normal Extended Logic Programs

Extended logic programs often fail to have an answer set. A similar problem arises in the context of Reiter's *default logic*. To deal with the problem of the potential non-existence of extensions in default logic, a possible solution is to restrict the syntax of knowledge representation. In [5], for instance, it is shown that a *normal default theory*, in which every default has the form $\frac{\alpha:\beta}{\beta}$, always yields at least one extension. In spite of their restricted syntax, normal default theories are useful to encode a large class of commonsense knowledge. An interesting question is then whether such an approach would also be feasible for logic programming. That is, can we state some possible restrictions on the syntax and content of an ELP, under which the existence of answer sets is guaranteed?

Analogously to default logic, one possible solution would be to restrict the use of NAF in defeasible rules. That is, we restrict default negation only to occur for a literal that is the opposite of the head of the same rule. More precisely, a defeasible rule of the form:

$$\mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{not}\ \neg\mathtt{c} \tag{3}$$

is called a *normal rule*. There is a good reason to call this type of rules normal. In fact, according to [3], the above normal rule is essentially the same as a normal default of the form: $\mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_n : \mathtt{c}\ /\ \mathtt{c}$.

Unfortunately, the mere restriction that all defeasible rules should be normal rules is not enough to guarantee the existence of answer sets. The "Married John" example illustrated in Section 1 is a counter-example of this restriction. One possible diagnosis of the "Married John" example is that, apparently, some information is missing. From our commonsense knowledge, we know that someone without a spouse is not married ($\neg\mathtt{m} \leftarrow \neg\mathtt{hs}$) and that someone who has a spouse is not a bachelor ($\neg\mathtt{b} \leftarrow \mathtt{hs}$). Notice that these two rules are actually contraposed forms of the existing rules $\mathtt{hs} \leftarrow \mathtt{m}$ and $\neg\mathtt{hs} \leftarrow \mathtt{b}$. Adding these rules yields the following logic program.

$$\begin{array}{lll@{\qquad}lll}
\mathtt{r} & \leftarrow & & \mathtt{hs} & \leftarrow & \mathtt{m} \\
\mathtt{p} & \leftarrow & & \neg\mathtt{m} & \leftarrow & \neg\mathtt{hs} \\
\mathtt{m} & \leftarrow & \mathtt{r}, \mathtt{not}\,\neg\,\mathtt{m} & \neg\mathtt{hs} & \leftarrow & \mathtt{b} \\
\mathtt{b} & \leftarrow & \mathtt{p}, \mathtt{not}\,\neg\,\mathtt{b} & \neg\mathtt{b} & \leftarrow & \mathtt{hs}\,.
\end{array}$$

The above program has two answer sets: $\{\mathtt{r}, \mathtt{p}, \mathtt{m}, \mathtt{hs}\}$ and $\{\mathtt{r}, \mathtt{p}, \mathtt{b}, \neg\mathtt{hs}\}$. This outcome can be seen as more desirable than the outcome of the original program, where no answer set exists.

Nevertheless, contraposition (or even *transposition*, as introduced in [2]) may not be enough to guarantee the existence of answer sets. Consider the following program:

$$\begin{array}{lll@{\qquad}lll}
\mathtt{a} & \leftarrow & & \mathtt{b} & \leftarrow & \mathtt{a}, \mathtt{not}\,\neg\,\mathtt{b} \\
\mathtt{c} & \leftarrow & \mathtt{b} & \neg\mathtt{b} & \leftarrow & \mathtt{c}\,.
\end{array}$$

In this program, all defeasible rules are normal, but even when one adds the rules $\neg\mathtt{b} \leftarrow \neg\mathtt{c}$ and $\neg\mathtt{c} \leftarrow \mathtt{b}$ (which makes the set of strict rules closed under contraposition) the program still does not yield any answer sets. It indicates that in order to guarantee the existence of answer sets, additional requirements are necessary.

Definition 3.1 (transpositive, transitive, antecedent-cleaned)
Let s_1 and s_2 be strict rules. We say that s_2 is a *transpositive* version of s_1 iff:
$s_1 = \mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n$ and
$s_2 = \neg\mathtt{a}_i \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{i-1}, \neg\mathtt{c}, \mathtt{a}_{i+1}, \ldots, \mathtt{a}_n$ for some $1 \leq i \leq n$.
Let s_1, s_2 and s_3 be strict rules. We say that s_3 is a *transitive* version of s_1 and s_2 iff:
$s_1 = \mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n$,
$s_2 = \mathtt{a}_i \leftarrow \mathtt{b}_1, \ldots, \mathtt{b}_m$ for some $1 \leq i \leq n$, and
$s_3 = \mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{i-1}, \mathtt{b}_1, \ldots, \mathtt{b}_m, \mathtt{a}_{i+1}, \ldots, \mathtt{a}_n$.
Let s_1 and s_2 be strict rules. We say that s_2 is an *antecedent cleaned* version of s_1 iff:
$s_1 = \neg\mathtt{a}_i \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_i, \ldots, \mathtt{a}_n$ and
$s_2 = \neg\mathtt{a}_i \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{i-1}, \mathtt{a}_{i+1}, \ldots, \mathtt{a}_n$.

The intuition behind transposition can be illustrated by translating a strict rule $\mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n$ to a material implication $\mathtt{c} \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_n$. This implication is logically equivalent to $\neg\mathtt{a}_i \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_{i-1} \wedge \neg\mathtt{c} \wedge \mathtt{a}_{i+1} \wedge \cdots \wedge \mathtt{a}_n$, which is again translated to $\neg\mathtt{a}_i \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{i-1}, \neg\mathtt{c}, \mathtt{a}_{i+1}, \ldots, \mathtt{a}_n$. Notice that, when $n = 1$, transposition coincides with classical contraposition. Transitivity, as used in Definition 3.1, basically boils down to the substitution of a literal in the body of a rule with the body of another rule that has this literal as its head. The meaning of antecedent cleaning is also straightforward. Translate a strict rule $\neg\mathtt{a}_i \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_i \ldots, \mathtt{a}_n$ to a material implication $\neg\mathtt{a}_i \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_i \wedge \cdots \wedge \mathtt{a}_n$, which is equivalent to $\neg\mathtt{a}_i \vee \neg\mathtt{a}_1 \vee \cdots \vee \neg\mathtt{a}_i \vee \cdots \vee \neg\mathtt{a}_n$. In this formula, the double occurrence of $\neg\mathtt{a}_i$ can be eliminated, yielding $\neg\mathtt{a}_i \vee \neg\mathtt{a}_1 \vee \cdots \vee \neg\mathtt{a}_{i-1} \vee \neg\mathtt{a}_{i+1} \vee \cdots \vee \mathtt{a}_n$, which is equivalent to $\neg\mathtt{a}_i \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_{i-1} \wedge \mathtt{a}_{i+1} \wedge \cdots \wedge \mathtt{a}_n$, and is translated back to $\neg\mathtt{a}_1 \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{i-1}, \mathtt{a}_{i+1}, \ldots, \mathtt{a}_n$.

Definition 3.2 (closed) Let S be a set of strict rules. Then,
(i) S is *closed under transposition* iff for each rule s_1 in S, a rule s_2 is in S if s_2 is a transpositive version of s_1.
(ii) S is *closed under transitivity* iff for each rule s_1 and s_2 in S, a rule s_3 is in S if r_3 is a transitive version of s_1 and s_2.
(iii) S is *closed under antecedent cleaning* iff for each rule s_1 in S, a rule s_2 is in S if s_2 is an antecedent cleaned version of s_1.

Definition 3.3 (normal ELP) A program P is called a *normal extended logic program* (normal ELP, for short) iff:
(1) $strict(P)$ is closed under transposition, transitivity and antecedent cleaning, and
(2) $defeasible(P)$ consists of normal rules only.

Theorem 1 *Any normal ELP P has at least one answer set.*

REFERENCES

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2002.

[2] M. Caminada and L. Amgoud. An axiomatic account of formal argumentation. In: *Proceedings of the 20th National Conference of Artificial Intelligence*, MIT Press, 2005.

[3] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9(3/4):365–385, 1991.

[4] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence* 138:39–54, 2002.

[5] R. Reiter. A logic for default reasoning, *Artificial Intelligence* 13:81–132, 1980.