On the Existence of Answer Sets in Normal Extended Logic Programs[*]

Martin Caminada
Institute of Information
and Computing Sciences
Utrecht University, The Netherlands
martinc@cs.uu.nl

Chiaki Sakama
Department of Computer
and Communication Sciences
Wakayama University, Japan
sakama@sys.wakayama-u.ac.jp

Abstract

An often problematic feature in *answer set programming* is that a program does not always produce an answer set, even for programs which represent default information in a seemingly natural way. 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; 10; 11]. In ASP a problem is represented as an *extended logic program* whose declarative meaning is given by the *answer set semantics* [9]. An often problematic feature in ASP is that relatively small pieces of information may cause a total absence of answer sets. For example, the existence of a single rule like "$\mathtt{p} \leftarrow \mathtt{not\,p}$" in a program blocks all useful inferences from other parts of the program.

From the perspective of knowledge representation, such a "negative loop" – a literal depending on its default negation – 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 are seemingly natural and 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 definition, has a spouse." (f) "A bachelor, by definition, does not have a spouse." These sentences are represented by the program P:

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

The program contains no negative loop, and encodes the given information in a natural way. The program has no answer set, however. It is interesting to observe that the problem does not arise if we encode the same knowledge in Reiter's *default logic* [13]. In default theories, (a) and (b) would become facts, (c) and (d) would become normal defaults, and (e) and (f) would become material implications. With this translation, the default theory D:

$$\mathtt{r}, \quad \mathtt{p}, \quad \mathtt{m} \supset \mathtt{hs}, \quad \mathtt{b} \supset \neg\,\mathtt{hs}, \quad \frac{\mathtt{r} : \mathtt{m}}{\mathtt{m}}, \quad \frac{\mathtt{p} : \mathtt{b}}{\mathtt{b}}.$$

has two extensions: one including $\{\mathtt{r}, \mathtt{p}, \mathtt{b}, \neg\mathtt{hs}, \neg\mathtt{m}\}$, the other including $\{\mathtt{r}, \mathtt{p}, \mathtt{m}, \mathtt{hs}, \neg\mathtt{b}\}$. In fact, the default theory D is a so-called *normal default theory* and always has an extension.

What is the cause of this difference? The point is that in extended logic programs a rule is interpreted as a one-way "inference rule". According to [9, page 367], "The language of extended programs includes classical negation, but not classical implication". As a result, a rule does not have a contrapositive meaning even if it is a definite rule without default negation. On the other hand, definite information in default logic is represented by first-order formulas having contrapositive meaning. To bridge the gap between extended logic programs and default theories, the paper [9] characterizes the extended logic program P in terms of the (non-normal) default theory:

$$\mathtt{r}, \quad \mathtt{p}, \quad \frac{\mathtt{m} :}{\mathtt{hs}}, \quad \frac{\mathtt{b} :}{\neg\,\mathtt{hs}}, \quad \frac{\mathtt{r} : \mathtt{m}}{\mathtt{m}}, \quad \frac{\mathtt{p} : \mathtt{b}}{\mathtt{b}}\,.$$

The above default theory has no extension either.

The fact that extended logic programs with normal default rules may not have any answer sets is somewhat discouraging. This is because normal default theories are "a very large and natural class of default theories" [13]. A knowledge engineer can encode problems like "Married-John" in a normal default theory in a straightforward manner, while he/she may fail to encode the same problem in a logic program. Our question is then: "How can one provide a natural meaning of an extended logic program with normal default rules?" or "Is there any condition to guarantee the existence of answer sets of extended logic programs with normal default rules?"

In this paper, we study the problem and propose a solution. We first introduce the class of *normal extended logic*

[*]This work has been supported by the EU ASPIC project.

programs whose default rules are in the form of normal defaults. We then consider some syntactic conditions to guarantee the existence of answer sets in such programs. We prove that under these conditions a normal extended logic program indeed always yields at least one answer set.

The rest of this paper is organized as follows. Section 2 introduces basic terms used in this paper. Section 3 discusses problems of ASP, and a solution for guaranteeing the presence of answer sets is given in Section 4. Section 5 verifies the usefulness of our approach in applications. In Section 6, we round off our discussion and related issues. Section 7 addresses conclusion as well as some open questions for future research.

2 Basic Definitions

A *program* considered in this paper is an *extended logic program* (ELP) [9], 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, namely, $\mathtt{a}$ or $\neg\mathtt{a}$ for an atom $\mathtt{a}$. $\mathtt{not}$ 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. The conjunction in the body is identified with the set of conjuncts in it. This means that a rule like $\mathtt{c} \leftarrow \mathtt{a}, \mathtt{a}$ is identified with its factored form $\mathtt{c} \leftarrow \mathtt{a}$. 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 with the empty body $\mathtt{c} \leftarrow$ is identified with the literal $\mathtt{c}$ and is called a *fact*. 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 variables. Throughout the paper, we handle finite ground programs unless stated otherwise.

The semantics of ELPs is given by the *answer set semantics* [9]. 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 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 The Problem

In nonmonotonic logics, a theory often fails to have an extension. To deal with the problem of the potential non-existence of extensions, a possible solution is to restrict the syntax of knowledge representation. In default logic, for instance, it is known 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:

$$\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, \mathtt{not}\, \neg\mathtt{c} \qquad (3)$$

is called a *normal rule*. There is a good reason to call this type of rule normal. In fact, according to [9], 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 warrant the existence of answer sets. The "Married John" example in Section 1 is a counter-example of this. 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.

Example 1. (Married John, continued)

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

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. Thus, one can see that at least in this example, adding the contraposed version of the strict rules solves the problem. Nevertheless, contraposition (or even *transposition*, as introduced in [2]) may not be enough to guarantee the existence of answer sets.

Example 2. Consider the program P_0:

$$\mathtt{a} \leftarrow, \qquad \mathtt{b} \leftarrow \mathtt{a},\, \mathtt{not}\, \neg\, \mathtt{b}, \qquad \mathtt{c} \leftarrow \mathtt{b}, \qquad \neg\mathtt{b} \leftarrow \mathtt{c}\,.$$

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 set.

One may consider that interpreting strict rules as *clauses* in propositional logic and representing strict rules as disjunctive facts would solve the problem. Rewriting strict rules as disjunctive facts in the above program, it becomes

$$\mathtt{a} \leftarrow, \quad \mathtt{b} \leftarrow \mathtt{a}, \mathtt{not}\,\neg\,\mathtt{b}, \quad \mathtt{c} \vee \neg\mathtt{b} \leftarrow, \quad \neg\mathtt{b} \vee \neg\,\mathtt{c} \leftarrow\ .$$

The modified disjunctive program has the single answer set $\{\,\mathtt{a}, \neg\mathtt{b}\,\}$. Unfortunately, such rewriting does not work in general. For instance, the following program, normal defeasible rules and disjunctive facts, has no answer set.[1]

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

The above discussion indicates that to warrant the existence of answer sets is not a simple problem, even if defeasible rules are restricted to normal default rules. In the next section, we investigate additional conditions to be put on strict rules.

4 Normal Extended Logic Programs

We introduce three closure operators on a set of strict rules.

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

Let s_1 and s_2 be strict rules. We say that s_2 is an *antecedent cleaned* version of s_1 iff:
$s_1 = \neg\mathtt{a}_\mathtt{i} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{i}, \ldots, \mathtt{a}_\mathtt{n}$ and
$s_2 = \neg\mathtt{a}_\mathtt{i} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{\mathtt{i}-1}, \mathtt{a}_{\mathtt{i}+1}, \ldots, \mathtt{a}_\mathtt{n}$.

The intuition behind transposition can be illustrated by translating a strict rule $\mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}$ to a material implication $\mathtt{c} \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_\mathtt{n}$. This implication is rewritten as a disjunction $\mathtt{c} \vee \neg(\mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_\mathtt{n})$, which in its turn can be written as a disjunction $\mathtt{c} \vee \neg\mathtt{a}_1 \vee \cdots \vee \neg\mathtt{a}_\mathtt{n}$. In this disjunction, different disjuncts can be put in front. Putting for instance $\mathtt{a}_\mathtt{i}$ in front yields $\neg\mathtt{a}_\mathtt{i} \vee \neg\mathtt{a}_1 \vee \cdots \vee \neg\mathtt{a}_{\mathtt{i}-1} \vee \mathtt{c} \vee \neg\mathtt{a}_{\mathtt{i}+1} \vee \cdots \vee \neg\mathtt{a}_\mathtt{n}$, which is equivalent to $\neg\mathtt{a}_\mathtt{i} \vee \neg(\mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_{\mathtt{i}-1} \wedge \neg\mathtt{c} \wedge \mathtt{a}_{\mathtt{i}+1} \wedge \cdots \wedge \mathtt{a}_\mathtt{n})$. This implication is logically equivalent to $\neg\mathtt{a}_\mathtt{i} \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_{\mathtt{i}-1} \wedge \neg\mathtt{c} \wedge \mathtt{a}_{\mathtt{i}+1} \wedge \cdots \wedge \mathtt{a}_\mathtt{n}$, which is again translated to $\neg\mathtt{a}_\mathtt{i} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{\mathtt{i}-1}, \neg\mathtt{c}, \mathtt{a}_{\mathtt{i}+1}, \ldots, \mathtt{a}_\mathtt{n}$. Notice that, when $n = 1$, transposition coincides with classical contraposition.

Transitivity 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 intuition behind antecedent cleaning can be illustrated by translating a strict rule $\neg\mathtt{a}_\mathtt{i} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{i} \ldots, \mathtt{a}_\mathtt{n}$ to a material implication $\neg\mathtt{a}_\mathtt{i} \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_\mathtt{i} \wedge \cdots \wedge \mathtt{a}_\mathtt{n}$, which is then equivalent to the disjunction $\neg\mathtt{a}_\mathtt{i} \vee \neg\mathtt{a}_1 \vee \cdots \vee \neg\mathtt{a}_\mathtt{i} \vee \cdots \vee \neg\mathtt{a}_\mathtt{n}$. In this formula, the double occurrence of $\neg\mathtt{a}_\mathtt{i}$ can be eliminated, yielding $\neg\mathtt{a}_\mathtt{i} \vee \neg\mathtt{a}_1 \vee \cdots \vee \neg\mathtt{a}_{\mathtt{i}-1} \vee \neg\mathtt{a}_{\mathtt{i}+1} \vee \cdots \vee \mathtt{a}_\mathtt{n}$, which is equivalent to $\neg\mathtt{a}_\mathtt{i} \subset \mathtt{a}_1 \wedge \cdots \wedge \mathtt{a}_{\mathtt{i}-1} \wedge\ \ \mathtt{a}_{\mathtt{i}+1} \wedge \cdots \wedge \mathtt{a}_\mathtt{n}$. This is then translated to the rule $\neg\mathtt{a}_\mathtt{i} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_{\mathtt{i}-1}, \mathtt{a}_{\mathtt{i}+1}, \ldots, \mathtt{a}_\mathtt{n}$.

Definition 2. Let S be a set of strict rules. Then,

(i) S is *closed under transposition* iff for each rule $s_1 \in S$, if s_2 is a transpositive version of s_1 then $s_2 \in S$.

(ii) S is *closed under transitivity* iff for each rule $s_1, s_2 \in S$, if r_3 is a transitive version of s_1 and s_2 then $s_3 \in S$.

(iii) S is *closed under antecedent cleaning* iff for each rule $s_1 \in S$, if s_2 is an antecedent cleaned version of s_1 then $s_2 \in S$.

Definition 3. A program P is called a *normal extended logic program* (normal ELP, for short) iff:

1. $\mathit{strict}(P)$ is closed under transposition, transitivity and antecedent cleaning, and
2. $\mathit{defeasible}(P)$ consists of normal rules only.

Example 3. The program P_0 of Example 2 becomes P_1 below by closing under transposition, transitivity, and antecedent cleaning:

$$\begin{array}{lllll} P_1: & \mathtt{a} \leftarrow\ , & \mathtt{b} \leftarrow \mathtt{a}, \mathtt{not}\,\neg\,\mathtt{b}, & \mathtt{c} \leftarrow \mathtt{b}, & \neg\mathtt{b} \leftarrow \mathtt{c}, \\ & \neg\mathtt{b} \leftarrow \neg\mathtt{c}, & \neg\mathtt{c} \leftarrow \mathtt{b}, & \neg\mathtt{b} \leftarrow \mathtt{b}, & \neg\mathtt{b} \leftarrow\ . \end{array}$$

Given a set S of strict rules, let $\mathit{Cl\_transposition}(S)$, $\mathit{Cl\_transitivity}(S)$, and $\mathit{Cl\_antclearning}(S)$ be closure computation under transposition, transitivity, and antecedent cleaning, respectively. Then, a closed set S under each closure computation is a set satisfying the following three conditions:

$$S = \mathit{Cl\_transposition}(S),$$
$$S = \mathit{Cl\_transitivity}(S),$$
$$S = \mathit{Cl\_antcleaning}(S).$$

A procedure for computing such a closed set S from a non-closed set S' of strict rules is given as follows.

```
Repeat
    S := S'
    S' := Cl_transposition(S');
    S' := Cl_transitivity(S');
    S' := Cl_antcleaning(S');
Until (S = S').
```

Within the above loop, all the operations $\mathit{Cl\_transposition}$, $\mathit{Cl\_transitivity}$, and $\mathit{Cl\_antcleaning}$, can be applied in any order (i.e., the final result $\mathtt{S} = \mathtt{S}'$ is always unique). Formally, we have the following result.

Proposition 1. *Given a (non-closed) set S' of strict rules, let S and T be any two closed sets that result from the above procedure by applying the three closure operations in some specific (possibly different) order. Then, $S = T$ holds.*

Proof. Let Cl_1, Cl_2 and Cl_3 be three different closure operators such that Cl_i is one of $\mathit{Cl\_transposition}$, $\mathit{Cl\_transitivity}$, and $\mathit{Cl\_antclearning}$. Since each iteration of the loop in the procedure monotonically increases S',

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

it holds that $S' \subseteq S$. The fact that the loop terminates means that S is closed under Cl_1, Cl_2 and Cl_3, namely, $Cl_1(S) = S$, $Cl_2(S) = S$, and $Cl_3(S) = S$. Suppose that the procedure produces a set T such that $S' \subseteq T$, $Cl_1(T) = T$, $Cl_2(T) = T$, $Cl_3(T) = T$, but $S \neq T$. In this case, there is some element that is in S but not in T, or in T but not in S. Assume without loss of generality that there is some element that is in S but not in T (i.e., $S \setminus T \neq \emptyset$). Suppose a sequence $S_0, \ldots S_k$ of sets of strict rules where $S_0 = S'$ and $S_k = S$. Let S_i $(0 \leq i \leq k)$ be the *first* set in the sequence that contains an element in $S \setminus T$. This implies that $S_{i-1} \subseteq T$. Assume that S_i is constructed from S_{i-1} by applying Cl_j, namely, $Cl_j(S_{i-1}) = S_i$. By the monotonicity of a closure operator, $S_{i-1} \subseteq T$ implies $Cl_j(S_{i-1}) \subseteq Cl_j(T)$. The equations $Cl_j(S_{i-1}) = S_i$ and $Cl_j(T) = T$ then imply that $S_i \subseteq T$. This contradicts the assumption that S_i contains an element in $S \setminus T$. □

By Proposition 1, given two programs P_1 and P_2 such that $strict(P_1) = strict(P_2)$ and $defeasible(P_1) = defeasible(P_2)$, the same normal ELP is constructed by closing P_1 and P_2 under three operations.

Our goal is to prove that a normal ELP always has at least one answer set. We first introduce some notions.

Definition 4. Let P be a program. An *entailment tree* (ET) under P is a finite non-empty tree satisfying the following:

1. each node of the tree is a strict rule from P, and
2. if a strict rule s_2 is a child node of a strict rule s_1, then the head of s_2 appears in the body of s_1. That is:

   $s_1 = \mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}$, and

   $s_2 = \mathtt{a_i} \leftarrow \mathtt{b_1}, \ldots, \mathtt{b_m}$ for some $1 \leq i \leq n$.

We say that an entailment tree ET *needs* a literal $\mathtt{a_i}$ iff ET contains a node $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}$ that has no child with $\mathtt{a_i}$ $(1 \leq i \leq n)$ as a head.

Definition 5. We say that a literal $\mathtt{c}$ *follows from* a set L of literals under a program P iff:

1. $\mathtt{c}$ is in L, or
2. there is an entailment tree ET, of which every node is in $strict(P)$, with $\mathtt{c}$ as the head of its root-node and where every literal $\mathtt{l}$ needed by ET is in L.

By definition, the set of strict rules $strict(P)$ is consistent iff there is no literal $\mathtt{p}$ such that both $\mathtt{p}$ and $\neg\mathtt{p}$ follow from $\emptyset$ under $strict(P)$.

Definition 6. Let P be a program and $[d_1, \ldots, d_n]$ $(n \geq 0)$ a list of defeasible rules from P. We define $Result_P([d_1, \ldots, d_n])$ as the set of all literals that follow from $\{head(d_1), \ldots, head(d_n)\}$ under P. We say that $Result_P([d_1, \ldots, d_n])$ is *consistent* if it does not contain both $\mathtt{p}$ and $\neg\mathtt{p}$ at the same time for any literal $\mathtt{p}$.

Definition 7. Let P be a program and $[d_1, \ldots, d_n]$ $(n \geq 0)$ a list of defeasible rules from P. We say that $[d_1, \ldots, d_n]$ is a *trace* of P iff:

1. for any i, j (s.t. $1 \leq i \leq n$ and $1 \leq j \leq n$ and $i \neq j$) it holds that $d_i \neq d_j$, and
2. for any $d_i \in \{d_1, \ldots, d_n\}$ with $d_i = \mathtt{q_i} \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_{m_i}}, \mathtt{not}\neg\mathtt{q_i}$, it holds that:
   (a) $\mathtt{p_1}, \ldots, \mathtt{p_{m_i}} \in Result_P([d_1, \ldots, d_{i-1}])$ and
   (b) $\neg\mathtt{q_i} \notin Result_P([d_1, \ldots, d_{i-1}])$.

In particular, a trace $[d_1, \ldots, d_n]$ is called *terminated* iff there is no $d_{n+1} \in defeasible(P)$ such that $[d_1, \ldots, d_n, d_{n+1}]$ is a trace of P.

Proposition 2. *Any normal ELP has at least one terminated trace.*

Proof. Such a trace is constructed by starting with the empty trace and successively adding defeasible rules until no defeasible rule can be added anymore. As a program is finite, every trace terminates. □

Example 4. Let P_1 be the program of Example 3. Then, $Result_{P_1}([\,]) = \{\mathtt{a}, \neg\mathtt{b}\}$ and $Result_{P_1}([\mathtt{b} \leftarrow \mathtt{a}, \mathtt{not}\,\neg\mathtt{b}]) = \{\mathtt{a}, \neg\mathtt{b}\}$, where $[\mathtt{b} \leftarrow \mathtt{a}, \mathtt{not}\,\neg\mathtt{b}]$ is a terminated trace.

In what follows, we write $Head_i$ for $\{head(d_1), \ldots, head(d_i)\}$.

Lemma 1. *Let $[d_1, \ldots, d_n]$ be a trace of a normal ELP P. If $strict(P)$ is consistent, $Result_P([d_1, \ldots, d_n])$ is consistent.*

Proof. Suppose $Result_P([d_1, \ldots, d_n])$ is inconsistent while $strict(P)$ is consistent. As $strict(P)$ is consistent, there must be some smallest (but non-empty) sublist $[d_1, \ldots, d_i]$ (with $1 \leq i \leq n$) where $Result_P([d_1, \ldots, d_i])$ is inconsistent. The fact that this sublist is the smallest means that $Result_P([d_1, \ldots, d_{i-1}])$ is consistent. The fact that $Result_P([d_1, \ldots, d_i])$ is inconsistent means, by Definition 6, that there is some literal $\mathtt{r}$ such that both $\mathtt{r}$ and $\neg\mathtt{r}$ follow from $Head_i$ under P. Given the fact that both $\mathtt{r}$ and $\neg\mathtt{r}$ follow from $Head_i$, we prove that this must be the case because of point 2 (and not point 1) of Definition 5. Suppose that this is not the case. Then point 1 of Definition 5 is the case for at least $\mathtt{r}$ or $\neg\mathtt{r}$. We distinguish three cases:

1. Both $\mathtt{r}$ and $\neg\mathtt{r}$ follow from $Head_i$ because of point 1 of Definition 5. In that case $\mathtt{r} \in Head_i$ and $\neg\mathtt{r} \in Head_i$. But then $[d_1, \ldots, d_i]$ would not be a trace, since it cannot contain both $\mathtt{r}$ and $\neg\mathtt{r}$. Contradiction.
2. $\mathtt{r}$ follows from $Head_i$ because of point 1 of Definition 5 and $\neg\mathtt{r}$ follows from $Head_i$ because of point 2 of Definition 5. In that case, $\mathtt{r} \in Head_i$ and there exists some entailment-tree ET for $\neg\mathtt{r}$ such that every literal $\mathtt{l}$ that ET needs is in $Head_i$. We distinguish two subcases:
   (a) ET does not need $head(d_i)$. In that case, because $strict(P)$ is closed under transitivity, $strict(P)$ contains a rule $\neg\mathtt{r} \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_n}$ where $\{\mathtt{s_1}, \ldots, \mathtt{s_n}\} \subseteq Head_{i-1}$. In that case, since $Result_P([d_1, \ldots, d_{i-1}])$ is assumed to be consistent, $\mathtt{r}$ must be equal to $head(d_i)$. But then $[d_1, \ldots, d_i]$ would not be a trace. Contradiction.
   (b) ET does need $head(d_i)$. In that case, because $strict(P)$ is closed under transitivity, $strict(P)$ contains a rule $\neg\mathtt{r} \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_n}, head(d_i)$ where $\{\mathtt{s_1}, \ldots, \mathtt{s_n}\} \subseteq Head_{i-1}$. Because $\neg\mathtt{r}$ is assumed to be $\neg head(d_j)$ (for some $1 \leq j \leq i$), this rule is actually $\neg head(d_j) \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_n}, head(d_i)$. As

$strict(P)$ is closed under transposition, there also exists a rule $\neg head(d_i) \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_n}, head(d_j)$. We distinguish two cases:

   i. $j = i$. In that case, because $strict(P)$ is closed under antecedent cleaning, there also exists a strict rule $\neg head(d_i) \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_n}$. But then $[d_1, \ldots, d_i]$ would not be a trace as $\neg head(d_i)$ follows from $Head_{i-1}$. Contradiction.

   ii. $j \neq i$. This can only be the case if $j < i$. In that case, every element of the body of $\neg head(d_i) \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_n}, head(d_j)$ is an element of $Head_{i-1}$. But then $[d_1, \ldots, d_i]$ would not be a trace as $\neg head(d_i)$ follows from $Head_{i-1}$. Contradiction.

3. $\mathtt{r}$ follows from $Head_i$ because of point 2 of Definition 5 and $\neg\mathtt{r}$ follows from $Head_i$ because of point 1 of Definition 5. This case goes similar to the preceding case.

As it is now proved that $\mathtt{r}$ and $\neg\mathtt{r}$ follow from $Head_i$ because of point 2 of Definition 5, it holds that both $\mathtt{r}$ and $\neg\mathtt{r}$ have entailment trees. Let ET_1 be the entailment-tree for $\mathtt{r}$ and ET_2 be the entailment-tree for $\neg\mathtt{r}$. We distinguish four cases:

1. Neither ET_1 nor ET_2 needs $head(d_i)$. In this case, $Result_P([d_1, \ldots, d_{i-1}])$ would also be inconsistent. Contradiction.

2. ET_1 needs $head(d_i)$ but ET_2 does not need $head(d_i)$. As $strict(P)$ is closed under transitivity, it contains the rule $\mathtt{r} \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_n}, head(d_i)$ from ET_1 and the rule $\neg\mathtt{r} \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_m}$ from ET_2, where each $\mathtt{p_j} \in Head_{i-1}$ $(1 \leq j \leq n)$ and each $\mathtt{s_j} \in Head_{i-1}$ $(1 \leq j \leq m)$. As $strict(P)$ is closed under transposition, there exists a rule: $\neg head(d_i) \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_n}, \neg\mathtt{r}$ in $strict(P)$. As $strict(P)$ is closed under transitivity, there exists a rule: $\neg head(d_i) \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_n}, \mathtt{s_1}, \ldots, \mathtt{s_m}$ in $strict(P)$. Now, the body of this rule contains literals exclusively from $Head_{i-1}$. This means that $\neg head(d_i)$ follows from $Head_{i-1}$. But this means that $[d_1, \ldots, d_i]$ would not be a trace. Contradiction.

3. ET_1 does not need $head(d_i)$ but ET_2 does need $head(d_i)$. This case is similar to the preceding case.

4. Both ET_1 and ET_2 need $head(d_i)$. As $strict(P)$ is closed under transitivity, it contains the following two rules: $\mathtt{r} \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_n}, head(d_i)$ from ET_1 and $\neg\mathtt{r} \leftarrow \mathtt{s_1}, \ldots, \mathtt{s_m}, head(d_i)$ from ET_2, where $\mathtt{p_j} \in Head_{i-1}$ $(1 \leq j \leq n)$ and $\mathtt{s_j} \in Head_{i-1}$ $(1 \leq j \leq m)$. As $strict(P)$ is closed under transposition, there exists a rule: $\neg head(d_i) \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_n}, \neg\mathtt{r}$ in $strict(P)$. As $strict(P)$ is closed under transitivity, there exists a rule: $\neg head(d_i) \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_n}, \mathtt{s_1}, \ldots, \mathtt{s_m}, head(d_i)$ in $strict(P)$. As $strict(P)$ is closed under antecedent cleaning, there exists a rule: $\neg head(d_i) \leftarrow \mathtt{p_1}, \ldots, \mathtt{p_n}, \mathtt{s_1}, \ldots, \mathtt{s_m}$ in $strict(P)$. Now, the body of this rule contains literals exclusively from $Head_{i-1}$. This means that $\neg head(d_i)$ follows from $Head_{i-1}$. But this means that $[d_1, \ldots, d_i]$ would not be a trace. Contradiction. □

Theorem 1. *Let $[d_1, \ldots, d_n]$ be a terminated trace of a normal ELP P. If $strict(P)$ is consistent, $Result_P([d_1, \ldots, d_n])$ is an answer set of P.*

Proof. When $strict(P)$ is consistent, $Result_P([d_1, \ldots, d_n])$ is consistent by Lemma 1. Let $S = Result_P([d_1, \ldots, d_n])$.

First of all, S satisfies P^S. Suppose this is not the case. Then there is some rule $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}$ in P^S such that $\{\mathtt{a_1}, \ldots, \mathtt{a_m}\} \subseteq S$ but $\mathtt{c} \notin S$. We distinguish two cases:

1. $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}$ is in $strict(P)$. By the definition of $Result_P([d_1, \ldots, d_n])$ and from the fact that $\{\mathtt{a_1}, \ldots, \mathtt{a_m}\} \subseteq S$, it follows that $\mathtt{c} \in S$. Contradiction.

2. $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}$ is generated as a reduct of a defeasible rule $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}, \mathtt{not}\ \neg\mathtt{c}$. This, by the definition of P^S, means that $\neg\mathtt{c} \notin S$. But then the trace $[d_1, \ldots, d_n]$ would not be terminated, as it would be possible to extend it with the defeasible rule $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}, \mathtt{not}\ \neg\mathtt{c}$. Contradiction.

Secondly, S is also a *minimal* set that satisfies P^S. Suppose that S was not minimal. Then there would be some $T \subsetneq S$ that also satisfies P^S. Let $U = S \backslash T$. Notice that U is not empty. Let $[d_1, \ldots, d_i]$ be the smallest sublist of $[d_1, \ldots, d_n]$ such that $Result_P([d_1, \ldots, d_i])$ contains an element $\mathtt{c} \in U$. We distinguish two cases:

1. $\mathtt{c}$ is in S because it is the head of some rule $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}, \mathtt{not}\ \neg\mathtt{c}$ in $[d_1, \ldots, d_n]$. As $[d_1, \ldots, d_i]$ is the smallest sublist such that $Result_P([d_1, \ldots, d_i])$ contains $\mathtt{c}$, this means that $d_i = \mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}, \mathtt{not}\ \neg\mathtt{c}$. By the definition of a trace, this means that $\{\mathtt{a_1}, \ldots, \mathtt{a_m}\} \subseteq Result_P([d_1, \ldots, d_{i-1}])$. As $[d_1, \ldots, d_i]$ is the smallest sublist such that $Result_P([d_1, \ldots, d_i])$ contains an element of U, it holds that $Result_P([d_1, \ldots, d_{i-1}])$ does not contain any element of U. Therefore, $Result_P([d_1, \ldots, d_{i-1}]) \subseteq T$. Therefore, $\{\mathtt{a_1}, \ldots, \mathtt{a_m}\} \subseteq T$. As S (and T) is consistent (by Lemma 1), it holds that $\neg\mathtt{c} \notin S$ (or T). Therefore, $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m} \in P^S$. As T is assumed to satisfy P^S and $\{\mathtt{a_1}, \ldots, \mathtt{a_m}\} \subseteq T$, this also means that $\mathtt{c} \in T$. Contradiction.

2. The preceding case is not applicable and $\mathtt{c}$ is in S because it is the head of some strict rule $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}$. Now, let's examine the entailment tree ET for $\mathtt{c}$. For the top-rule of ET, which is $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_m}$, it holds that $\mathtt{c}$ is not in T, so from the fact that T satisfies $strict(P)$, it follows that there must be some $\mathtt{a_i}$ $(1 \leq i \leq m)$ such that $\mathtt{a_i} \notin T$. Now, let's look at the subtree for $\mathtt{a_i}$. It contains a top-rule $\mathtt{a_i} \leftarrow \mathtt{b_1}, \ldots, \mathtt{b_k}$, for which it holds that $\mathtt{a_i}$ is not in T, so from the fact that T satisfies $strict(P)$ it follows that there must be some $\mathtt{b_j}$ $(1 \leq j \leq k)$ such that $\mathtt{b_j} \notin T$. So, basically, the entailment tree ET contains a path of strict rules of which the head is not in T and at least one literal in the body is not in T. When this path ends, it does so with a strict rule $\mathtt{e} \leftarrow \mathtt{f_1}, \ldots, \mathtt{f_l}$ where each $\mathtt{f_h} (1 \leq h \leq l)$ is in $Head_{i-1}$. But $Head_{i-1} \subseteq T$. This is because $Result_P([d_1, \ldots, d_{i-1}])$ does not contain an element of U, as $[d_1, \ldots, d_i]$ is the smallest sublist such that $Result_P([d_1, \ldots, d_i])$ contains an element of U. But then T does not satisfy the strict rule $\mathtt{e} \leftarrow \mathtt{f_1}, \ldots, \mathtt{f_l}$. Contradiction. □

Theorem 2. *Any normal ELP P has at least one answer set.*

Proof. When $strict(P)$ is consistent, P has a terminated trace $[d_1, \ldots, d_n]$ (Proposition 2), and $Result_P([d_1, \ldots, d_n])$ becomes an answer set of P (Theorem 1). Else when $strict(P)$ is inconsistent, P has the answer set Lit. $\square$

Example 5. P_1 in Example 4 has the answer set $\{\mathtt{a}, \neg\mathtt{b}\}$.

5 Application

Normal ELPs restrict defeasible rules to normal ones. Nevertheless, they are still useful for representing many interesting problems. For instance, a number of *constraint satisfaction problems* (CSPs) are represented using normal rules in answer set programming. The following example is due to [11].

Example 6. Put N pigeons into M holes so that there is at most one pigeon in a hole. This problem is coded in the program:

$$\begin{aligned}
&\mathtt{pos(P,H)} \leftarrow \mathtt{pigeon(P)},\, \mathtt{hole(H)},\, \mathtt{not}\,\neg\,\mathtt{pos(P,H)},\\
&\neg\,\mathtt{pos(P,H)} \leftarrow \mathtt{pigeon(P)},\, \mathtt{hole(H)},\, \mathtt{not}\,\mathtt{pos(P,H)},\\
&\leftarrow \mathtt{pigeon(P)},\, \mathtt{hole(H)},\, \mathtt{hole(H')},\\
&\qquad\qquad \mathtt{pos(P,H)},\, \mathtt{pos(P,H')},\, \mathtt{H} \neq \mathtt{H'},\\
&\leftarrow \mathtt{pigeon(P)},\, \mathtt{not}\,\mathtt{hashole(P)},\\
&\mathtt{hashole(P)} \leftarrow \mathtt{pigeon(P)},\, \mathtt{hole(H)},\, \mathtt{pos(P,H)},\\
&\leftarrow \mathtt{pigeon(P)},\, \mathtt{pigeon(P')},\, \mathtt{hole(H)},\\
&\qquad\qquad \mathtt{pos(P,H)},\, \mathtt{pos(P',H)},\, \mathtt{P} \neq \mathtt{P'},
\end{aligned}$$

where `hole` and `pigeon` give the available holes and pigeons, and $\mathtt{pos(P,H)}$ represents a legal position of a pigeon $\mathtt{P}$ in a hole $\mathtt{H}$.

A careful reader may notice that the above program is beyond the class of normal ELPs, even after closing the strict rule under three operations, due to the existence of *integrity constraints*. By definition, normal ELPs do not contain integrity constraints – rules with with empty heads. In fact, an integrity constraint of the form: $\leftarrow \mathtt{F}$ where $\mathtt{F}$ is a conjunction of literals and NAF-literals, is semantically equivalent to the defeasible rule of the form: $\mathtt{c} \leftarrow \mathtt{F}, \mathtt{not}\,\mathtt{c}$ for any literal $\mathtt{c}$, under the answer set semantics. The defeasible rule presented above is not a normal rule, so that there is a reason to exclude integrity constraints from normal ELPs. In ASP, however, integrity constrains play an important role; imposing conditions on the solutions or pruning unwanted answer sets [11].

To manage integrity constrains in the context of normal ELPs, we handle them separate from a program. Formally, a set IC of integrity constrains is a set of rules of the form:

$$\leftarrow \mathtt{a_1},\, \ldots,\, \mathtt{a_n},\, \mathtt{not}\,\mathtt{b_1},\, \ldots,\, \mathtt{not}\,\mathtt{b_m}$$

where each $\mathtt{c}$, $\mathtt{a_i}$ and $\mathtt{b_j}$ is a positive/negative literal. Then, the following result holds.

Proposition 3. *Let P be a normal ELP and IC a set of integrity constraints. Then, if S is an answer set of $P \cup IC$, then S is an answer set of P.*

Proposition 3 presents that given a normal ELP P and a set IC of constrains, $P \cup IC$ eliminates answer sets of P which do not satisfy the constraints. If $P \cup IC$ has no answer set, this implies that P has no answer set satisfying IC. Thus, we can handle integrity constrains in normal ELPs as well. Normal rules plus integrity constrains can naturally encode many CSP problems and combinatorial problems such as the n-queen problems, the k-colorability problem, etc [11].

Normal ELPs have some advantage for providing a *collective* semantics for multiple knowledge bases. Suppose a *multi-agent system* which consists of k-agents. An agent i has a knowledge base P_i $(1 \leq i \leq k)$ in ASP, where definite knowledge is shared by every agent $strict(P_1) = \cdots = strict(P_k)$ (e.g., a shared ontology), while individual agents have their own default knowledge as $defeasible(P_i)$. In this situation, combining knowledge bases $\bigcup_{i=1}^{k} P_i$ may produce no answer set, even if each individual program P_i is consistent and meaningful. This is a serious problem of ASP in developing a large knowledge base or cooperative problem solving in multi-agent systems. When every P_i is coded in normal ELPs, the problem does not arise. The combined program $\bigcup_{i=1}^{k} P_i$ is also a normal ELP, so that it produces an answer set. The produced answer set is guaranteed to be consistent as far as the shared definite knowledge is consistent (because an ELP P has the answer set Lit iff $strict(P)$ has the answer set Lit).

Normal ELPs also have an advantage in program development. In ASP, a piece of information introduced to a program may lead to a total collapse of all entailment (no answer sets). This is a serious drawback when dealing with knowledge bases that evolve with time and change dynamically. In normal ELPs, on the other hand, such a "total collapse" problem never happens. A knowledge engineer feels free to update a part of a program without spoiling the global meaning as far as defeasible information is encoded in the form of normal rules.

6 Discussion

As for the problem of the potential absence of answer sets, one can distinguish two approaches. The first approach would be to change the semantics of logic programming. The use of the *well-founded semantics* and its variants [6], or the family of *paraconsistent semantics* [5] are of this kind. It would also be akin to the field of *formal argumentation*, where stable semantics has been changed for preferred or grounded semantics [7]. The second approach would be not to change the semantics but instead to syntactically restrict the form of knowledge that can be put into a knowledge base. It is this second approach that has been explored in this paper.

The idea of restricting the syntax of logic programs in order to warrant the existence of answer sets is not entirely new. For instance, Fages [8] and Costantini [4] investigate sufficient conditions for the existence of stable models of normal logic programs. However, a normal logic program contains no negative literal, so that it contains no rule of the form $\mathtt{c} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, \mathtt{not}\,\neg\mathtt{c}$. Thus, the result of this paper is not subsumed by preceding studies under the stable model semantics. Conversely, the class of normal ELPs does not cover the class of normal logic programs having stable models, such as *call-consistent programs* or *locally stratified programs*. In fact, such programs always have an answer set (or a stable model) and there is no need for closing the strict rules in those programs.

Syrjänen [14] develops a debugger in ASP. The debugger can detect inconsistencies stemming from negative-loops, but it cannot detect inconsistencies in a program like the "Married-John" example. Van Nieuwenborgh and Vermeir [12] propose a program transformation in which *every* rule of an ELP would become normal (i.e., $\mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{not}\, \neg\mathtt{c}$). This approach is in fact a special case of the approach put forward in this paper. By making all rules normal (and thus defeasible) the resulting program does not contain any strict rules. As an empty set of strict rules is trivially closed under transposition, transitivity and antecedent-cleaning, the result is by definition a normal ELP (in the sense of Definition 3). However, transforming every strict rule to a defeasible rule would be a very high price for warranting the existence of answer sets. In real life, one can find definite knowledge that always hold without exception. For instance, a human being is a mammal, a natural number is a rational number, etc. Strict rules represent such persistent information and, from the knowledge representation viewpoint, they should not be replaced by defeasible rules.

In our proposed solution, strict rules are supposed to be closed under transposition, transitivity, and antecedent cleaning. Among them, transitivity preserves answer sets of ELPs when executed as partial evaluation [6]. On the other hand, transposition and antecedent cleaning increase the entailment power of ASP in general. For instance, consider the program:

$$\mathtt{p} \leftarrow, \quad \neg\mathtt{p} \leftarrow \neg\mathtt{q}, \quad \mathtt{r} \leftarrow \neg\mathtt{r}\,.$$

The program has a single answer set $\{\mathtt{p}\}$, while transposition and antecedent cleaning produce additional consequences $\mathtt{q}$ and $\mathtt{r}$. Strengthening the entailment power of a logic may cause a problem such that hidden conflicts and inconsistencies can suddenly be entailed. For instance, the program

$$\mathtt{p} \leftarrow, \quad \neg\mathtt{p} \leftarrow \mathtt{q}, \quad \neg\mathtt{p} \leftarrow \neg\mathtt{q}\,.$$

has the consistent answer set $\{\mathtt{p}\}$, but introducing contraposition makes the program inconsistent. Such a conflict, in our view, in fact already existed, but was left concealed because entailment was too weak.

In ASP, strict rules do not have contrapositive meaning. This makes the theory of logic programming different from first-order logic. For instance, suppose an agent is having the information $P = \{\mathtt{p} \leftarrow \mathtt{q}\}$. When a new information $Q = \{\neg\mathtt{p}\}$ has arrived, which conclusion should be derived by the agent? If P is read as a Horn logic program, $\neg\mathtt{q}$ is derived from $P \cup Q$ because it is logically equivalent to $\{\mathtt{p} \vee \neg\mathtt{q},\ \neg\mathtt{p}\}$. If P is read as an extended logic program, by contrast, $\neg\mathtt{q}$ is not derived from $P \cup Q$. Thus, the same program may have different meaning depending on its reading, even in the absence of defeasible rules. In default logic, on the other hand, defeasible rules are represented by defaults, and strict rules are represented by first-order formulas. Comparing these two frameworks, default logic appears relatively simple, intuitive and straightforward to understand the principle of knowledge representation. Normal ELPs proposed in this paper have the same spirit as default logic in this sense. They can naturally encode normal default theories in logic programming, and enhance the use of ASP in knowledge representation.

From the perspective of computational complexity, computing a closure under transitivity, transposition and antecedent-cleaning introduces additional costs. In particular, transitivity would exponentially increase the number of strict rules in a program. The problem is, however, much related to the structure of a program. For instance, compare two programs:

$$\begin{aligned} P_2 :\quad & \mathtt{a} \leftarrow, \\ & \mathtt{b} \leftarrow \mathtt{a}, \\ & \mathtt{c} \leftarrow \mathtt{b}, \mathtt{not}\, \neg\mathtt{c}, \\ & \mathtt{d} \leftarrow \mathtt{c}, \\ & \mathtt{e} \leftarrow \mathtt{d}, \mathtt{not}\, \neg\mathtt{e}, \\ & \mathtt{f} \leftarrow \mathtt{e}; \end{aligned}$$

$$\begin{aligned} P_3 :\quad & \mathtt{a} \leftarrow, \\ & \mathtt{b} \leftarrow \mathtt{a}, \mathtt{not}\, \neg\mathtt{b}, \\ & \mathtt{c} \leftarrow \mathtt{b}, \\ & \mathtt{d} \leftarrow \mathtt{c}, \\ & \mathtt{e} \leftarrow \mathtt{d}, \\ & \mathtt{f} \leftarrow \mathtt{e}, \mathtt{not}\, \neg\mathtt{f}. \end{aligned}$$

In P_3 the cost of computing the closure of the strict rules is significantly higher than that of P_2, even though both programs have the same number of strict rules. As observed in the example, the exponential blow-up of closure computation would be caused not by the number of the all strict rules in a program, but by the number of "related" strict rules. If one could divide the strict rules into disjoint sets $S_1, \ldots, S_n$ such that there is no atom occurring both in a rule in S_i and in a rule in S_j $(i \neq j)$, then the maximal complexity is exponential with respect to the size of the largest S_i. So, even if a program P contains a large number of strict rules, the entire cost of closing the strict rules does not have to be huge, as long as the largest S_i is relatively small compared to the size of $\mathit{strict}(P)$. Computing transitivity is popularly done as partial evaluation in logic programming. In practice, the entire closure computation could be automatically done as a compilation process. Alternatively, in runtime environments the closure computation could be done only for the part of strict rules that are needed to solve a particular problem. The issue of how the additional rules of this closure could be generated dynamically when needed, is an interesting topic for future research. As normality of defeasible rules are known by their syntax, the task of checking whether an ELP is normal or not is equivalent to checking whether the strict part of the program is closed or not. Complexity for reasoning tasks in normal ELPs is the same as the one in ELPs.

Finally, it is worth mentioning that in the context of the well-founded semantics a *semi-normal* extended logic program, which consists of a set of defeasible rules of the form

$$\mathtt{c} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n,\ \mathtt{not}\ \mathtt{b}_1, \ldots,\ \mathtt{not}\ \mathtt{b}_m,\ \mathtt{not}\ \neg\mathtt{c}$$

and a set of strict rules that are closed under transposition, always has a consistent well-founded model [3]. In contrast to the present work, the study shows that it requires weaker conditions on the strict part and the defeasible part of a program. Detailed comparison and connection to the present work are topics for future research.

7 Summary

The potential absence of answer sets for extended logic programs is a serious problem that justifies efforts to find solutions for it. In this paper, we have examined the approach of restricting extended logic programs to be of a particular form. We have shown that for a normal extended logic program, where defeasible rules are normal and strict rules are closed under transposition, antecedent cleaning and transitivity, there always exists at least one answer set. This can be seen as similar to the field of default logic, where normal default theories also have at least one extension. In fact, it turns that there is a close relation between normal ELPs and normal default theories. A formal discussion on this connection will be reported elsewhere. An extension of the proposed framework to programs possibly containing disjunctions is another issue to be investigated.

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: *Proc. AAAI-05*, pp. 608–613, AAAI Press.

[3] M. Caminada. Well-founded semantics for semi-normal extended logic programs. In: *Proc. 11th Int'l Workshop on Nonmonotonic Reasoning*, Clausthal University of Technology, TR IfI-06-04, pp. 103–108, 2006.

[4] S. Costantini. On the existence of stable models. *Theory and Practice of Logic Programming* 6(1/2):169–212, 2006.

[5] C. V. Damásio and L. M. Pereira. A survey of paraconsistent semantics for logic programs. In: *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, vol. 2, D. M. Gabbay and Ph. Smets (eds.), Kluwer Academic, pp. 241–320, 1998.

[6] J. Dix. A classification theory of semantics of normal logic programs: weak properties. *Fundamenta Informatica* 22(3):257-288, 1995.

[7] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence* 77:321–357, 1995.

[8] F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science* 2:51–60, 1994.

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

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

[11] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*:25(3/4): 241–273, 1999.

[12] D. Van Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. *Proc. 8th European Conference on Logics in Artificial Intelligence*, LNAI 2424, pp. 432–443, Springer, 2002.

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

[14] T. Syrjänen. Debugging inconsistent answer set programs. In: *Proc. 11th International Workshop on Nonmonotonic Reasoning*, Clausthal University of Technology, TR IfI-06-04, pp. 77–83, 2006.